Software Abstractions for Parallel Architectures

Joel Falcou

Université Paris Sud
Spécialité : INFORMATIQUE

présentée par

Joel Falcou

pour obtenir:

L'HABILITATION À DIRIGER DES RECHERCHES DE L'UNIVERSITÉ PARIS SUD

Sujet de la thèse:

Abstractions Logicielles pour Architectures Parallèles

Software Abstractions for Parallel Architectures

soutenu le 1er Decembre, 2014

Jury :

Rapporteurs :	COLE Murray	Professeur	Université d'Edimburgh
	HAINS Gaetan	Professeur	Université Paris-Est Creteil
	HILL David	Professeur	Université Blaise Pascal
Examinateurs :	CONCHON Sylvain	Professeur	Université Paris 11
	MCCOLL Bill	Professeur	Université de Oxford HUAWEI Research Center
	ETIEMBLE Daniel	Professeur	Université Paris 11
	LAMOTTE Jean-Luc	Professeur	Université Paris 6
	VIALLE Stephane	Professeur	SUPELEC Metz

Thèse préparée à l'Université Paris-Sud au sein du Laboratoire de Recherche en Informatique (LRI), UMR 8623 CNRS et de l'équipe POSTAL, INRIA Saclay Île-de-France

Contents

Acknowledgments

This thesis is the story of 8 years of research work in various places with many people. It's important to understand that such a body of work can't realistically be done alone. I want to say thanks to every collegues and students that I encountered during those years : Lionel Lacassagne, Claude Tadonki, Tarik Saidani, Adrien Bak, Eric Jourdanneau, Khaled Hamidouche, Pierre Esterie, Alexandre Borghi, Cecile Germain, Sylvain Peyronnet, Thomas Herault, Marc Baboulin, Mikolaj Szydlarski, Sebastian Schaetz.Romain Montagne, Adrien Remy, Amal Khabou, Yushan Wang, Ian Masliah, Lenaic Bagnieres, Florence Laguzet, Dimitrios Chasapsis and Antoine Tran Tan. I also want to thanks people from NumScale for sharing this adventure with me: Mathias Gaunard, Alan Kelly, Charles Pretot, Thierry Mourrain and Guillaume Quintin.

My thanks also go to all my friends from the C++ Community that supported me and my Crazy Frenchman personna : Eric Niebler, Hartmut Kaiser, Bryce Lelbach, Joel de Guzman, Edouard Alligand, Michael Caisse and Andrew Sutton.

I want to express my gratitude to my reviewers – Pr Hill, Pr Hains and Pr Cole – for their precious time spent reading this thesis and for their extremely interesting remarks on my work. A great thanks to Pr Conchon, Pr Vialle, Pr McColl and Pr Lamotte for being part of my committee and for all the advices and discussions we exchanged.

I also want to thank Daniel Etiemble for everything since 2007. Thanks to have put your faith in me and to have been a model for me all those years. I won't probably there if you didn't decide to review my PHD thesis 8 years ago. Thanks again for your trust and support.

Finally, I want to thank my familly, my wife and my children to have been more than supporting during all those years. I love you.

Chapter 1

Introduction

Contents

> *"It is a profound and necessary truth that the deep things in science are not found because they are useful; they are found because it was possible to find them."*
>
> — J. Robert Oppenheimer

1.1 From Experiments to Simulations

Science has been built for centuries on top of a set of simple rules often called the Scientific Method. The scientific method is a four steps process that structure scientific reasoning. Those steps are usually defined as:

- the **Observation** of the subject of inquiry;
- the formulation of **Hypotheses** – or theory – to explain said observations;
- the definition of **Predictions**, including logical deduction from the theory;
- the design and execution of **Experiments** to validate all of the above.

This process has been driving scientific discovery and technological breakthrough for centuries. But as science ventured deeper into the understanding of phenomena which duration and scale were out of reach or for which reproducibility was an issue, the notion of experiments changed. Indeed, if setting up experiments to validate hypothesis on how heat transfers between macroscopic bodies can be made at human scale, how can we experiment the theory on the inner workings of a star, the first femtosecond of the Universe or the monitoring of generations of human beings ?

With the advent of computing and computer science, experiments evolved into **Numerical Simulations**. Such simulations are an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables in the simulation, predictions may be made about the behavior of the system and reproduce a large variety of settings, thus enabling an iterative refinement of hypotheses. Lately, computer simulations have also been a great tool to process the extremely large dataset that actual experiments can generate [Chen 2014].

Nowadays, numerical simulations running on computers is the most fundamental tool that most sciences – from physics to social science – use as a substitute to experiments when said experiments can not realistically be run with a satisfactory duration, budget or ethical framework. This also means that the accuracy and the speed at which such computer simulations can be done is a crucial factor for the global scientific advancement. If accuracy of the simulation is tied to the field knowledge of scientists, the speed of a simulation is tied to the way one may take advantage of a computer hardware.

1.2 The Free Lunch is Definitively Over

For many years, the computing power of a given computer was modeled as a direct derivation of the famous – yet often misquoted – Moore's Law. Moore's law is the observation that, over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years [Schaller 1997]. Since its initial definition in 1965, Moore's Law has been proven as accurate, in part because the law is now used in the semiconductor industry to guide long-term planning and to set targets for research and development. For years the self-fulfilling Moore's Law also dictated the increase of computing power of CPUs. A more popular version of Moore's Law, actually stated by David House, is that computing power doubles every 18 months. This phenomenon, often known as the *"Free Lunch"* following the term coined by Herb Sutter [Sutter 2005], has been true for more than four decades but as been recently been halted.

During the *"Free Lunch"* era, the computing power of microprocessors has resulted of the continuous increase of clock frequency and from micro-architectural features like super-scalar execution, caches hierarchy, branch predictor or SIMD extensions. However, the so called *"Heat Wall"* made high frequency unreachable while the benefits from micro-architectural features started showing diminishing returns effects. CPU manufacturers then started designing chips containing multiple computing cores as it was their only solution to the ever growing demand in microprocessor power. The so-called multi-cores then lead the path to many-cores chips containing hundreds of cores with far less micro-architectural features. As **parallelism** grew inside chips, the *"Free lunch"* was over as the average developers had

to willfully manage the complexity of parallel programming if they wanted to use a non trivial amount of performance from their hardware.

1.3 Challenges Addressed in this Document

In this context, current research directions usually focus either on how to provide ways for computer scientists to access the latent performance of modern complex parallel systems or on how to actually abstracts away hardware details so development productivity can be increased despite the complexity of parallel programming. In a more formal way, we struggle between **Abstractions**, *i.e* the ability to express parallelism with minimum concerns for implementation details, and **Efficiency** , *i.e* the ability to produce code whose performances can stay on the par with those obtained using low-level technologies. Those two research directions rarely deal with each other, leading to abstractions with interesting properties but no efficient implementation or to efficient tools being under-used due to a lack of proper user-level abstractions. Our work during the past six years focused on exploring how to design tools for parallel programming that provide a high level of abstractions while delivering a high level of performance. In addition to this classical conflict, we wanted to:

- Provide a **Non-Disruptive Technology** by avoiding the design of yet another language and associated ecosystem as we thought that the acceptance of our solutions, especially in industrial contexts, will be easier if our tools do not require changes in the programming infrastructure of our users. We choose to focus on the design of C++ software libraries so that their adoption can be maximal among computer scientists.
- Use **Domain Specific Knowledges** to drive optimizations. If low-level code generation quality is a given, the best opportunities of massive speed-up often lies in a precise choice of algorithms or algorithm's parameters that Domain knowledge may carry.
- Be **Architecture Aware**, by allowing our software solutions to support or be easily extensible to support upcoming parallel architectures. We first focused on small scale SIMD shared memory systems and moved onto clusters and accelerators.

Our work was then split into three phases:

Feasibility: Complex programming models can be implemented using various paradigms ranging from functional to object-oriented programming. Alas, the choice of the proper idiom and the proper language can be non-trivial. We explored if and how modern C++ design strategies like Generic Programming, Generative Programming and Template Meta-programming were able to solve the abstraction vs efficiency conundrum by constructing abstractions with few runtime penalties at

the cost of a more complex development process.

These results notably involved the supervision of Khaled Hamidouche, in collaboration with Daniel Etiemble.

Extensibility : Generic and Generative Programming are well known techniques for sequential development but applications to parallel programming suffered for a lack of generality in how the architectural component of the problem was handled. To solve this issue, we investigated an extension of those techniques to exploit architecture knowledge in a way compliant with Generic Programming. By relying on the properties of specific parallel programming models, we proposed a hierarchical version of the usual Generative Programming idiom.

These results notably involved the supervision of Pierre Esterie in collaboration with NumScale SAS.

Portability : Once our system was designed to support a wide selection of architectures and domains, we challenged its design by extending its support to different architectures and programming models.

These results notably involved the supervision of Antoine Tran Tan and Ian Masliah in collaboration with Daniel Etiemble, Lionel Lacassagne, Marc Baboulin and NumScale SAS.

1.4 Habilitation Thesis Overview

This document is organized as follows. Chapter 2 describes various abstractions for parallel programming. It describes how different approaches have been proposed to scale down the complexity of parallel applications by providing simplified execution and programming models for parallel systems. Chapter 3 presents modern software design principles and how they can help design proper implementation of parallel programming abstractions. Those methods are assessed in Chapter 4 by implementing a C++ BSP library based on generic programming. This implementation show how far we can go in term of both API expressiveness and performances. Based on these preliminary experiments, Chapter 5 depicts our efforts on providing scalable techniques for using architectural informations into the design of efficient libraries in a structured way. Chapter 6 and Chapter 7 shows how this improved design techniques were used into the design of various libraries using different sets of parallel programming abstractions while delivering a high level of performances. We then conclude on the contributions of this work and details some future research directions.

Chapter 2

Abstractions for Parallel Programming

Contents

> *"It is not only the violin that shapes the violinist, we are all shaped by the tools we train ourselves to use, and in this respect programming languages have a devious influence: they shape our thinking habits."*
> — Edsger W. Dijkstra

2.1 Motivation

The design of parallel algorithms is a complex process in which one should find a hardware-agnostic way to express the distribution, scheduling and potential synchronizations of an arbitrary number of sub-tasks applied on an arbitrary amount of data sets. Abstract models of parallel machines and parallel programs have been proposed to simplify the process of designing parallel algorithms. Those limited models willingly omit to take every aspect of parallel machines so that the general form of a program written within the model is simple. We can roughly divide those models in three great families based on the main aspect of parallel systems they focus on.

2.2 Performance Centric Abstraction

A first class of parallel programming models attempts to simplify the design of parallel programs by constraining valid programs to follow a set of rules tied to a runtime performance model. The objective is to provide a framework in which all valid programs performance can be analytically evaluated before any actual implementation. Even if other models may provide some sort of performance model, it is not usually as central. The relationship between most of these performance centric model is given in figure 2.1.

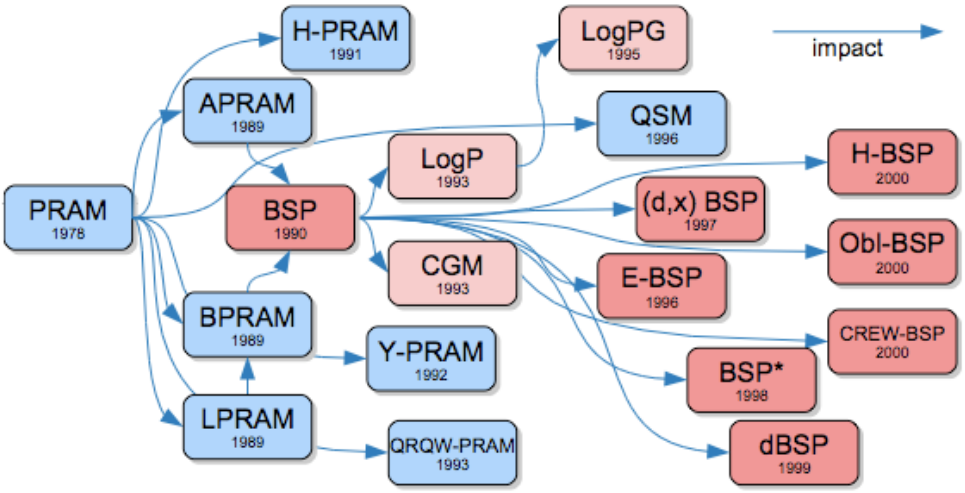

Figure 2.1: Relationships between performance centric parallel programming models

2.2.1 P-RAM Models

P-RAM [Fortune 1978] is a parallel computing model that is a direct extension of the classic sequential programming model. In the P-RAM model, P processors read and write in arbitrary locations of a global shared memory. Using a global clock, they synchronously execute their instructions. Communications between processors is handled by read/write patterns through the shared memory. Conflicts during memory accesses is then resolved using various strategies. The cost model of P-RAM is rather simple as it doesn't take synchronization or communication costs. P-RAM algorithm cost model is often expressed as depending only on the problem size and P-RAM system size.

Various extensions of P-RAM have been proposed:

- **Local-memory PRAM** (LP-RAM) has been proposed [Aggarwal 1989] as a way to exploit memory hierarchy inside a P-RAM machine. In addition to the global shared memory, each processor contains a private local memory. At each

step, each processor can either read from memory or perform computations. In this model, the concurrent accesses to the global memory are handled via the CREW model (Concurrent Read, Exclusive Write) to limit locking.

- **Block-PRAM** [Aggarwal 1990] extends LP-RAM by modeling the global memory as a series of blocks. Processors can then transfer data between global and local memory by block of consecutive cells. This block based access has a cost defined as $b + l$ where b is the block size and l the memory access latency, which is a machine dependent parameter. Accesses to block must be non-overlapping and are resolved using the EREW (Exclusive Read, Exclusive Write) model.

- **Asynchronous P-RAM** [Gibbons 1989] is a P-RAM extension featuring both local and global memory but, contrary to all P-RAM models, allowing asynchronous computation across processors. This model proposes a new instruction to force arbitrary synchronization among a group of processors and adds this synchronization cost to the performance model.

2.2.2 LOG-P Models

The LOG-P model proposed by [Culler 1993] is a model for distributed memory multi-processor machines using peer-to-peer message passing communications. The model models the machine based on a set of four characteristics:

- L: the network latency upper-bound for transmitting a word between two processors;

- o: the communication overhead, defined as the time required by a processor to complete a transmission and during which no computations can be performed;

- g: the delay between transmission of successive messages. Inverse of g can then be defined as the system bandwidth;

- P: the number of processors in the system.

Note that LOG-P models the network performance but has no informations about the network topology. Moreover, the network is supposed to have a finite capacity so that only L/g messages can be in transit at any given time. If a processor tries to transmit a message under these circumstances, it will be blocked until the network is available again.

2.2.3 BSP Models

The Bulk Synchronous Parallel Model (BSP) was introduced by [Valiant 1990] as a bridge between the hardware and the software to simplify the development of parallel algorithms. Classically, the BSP model is defined by three components: a machine model, a programming model and a cost model:

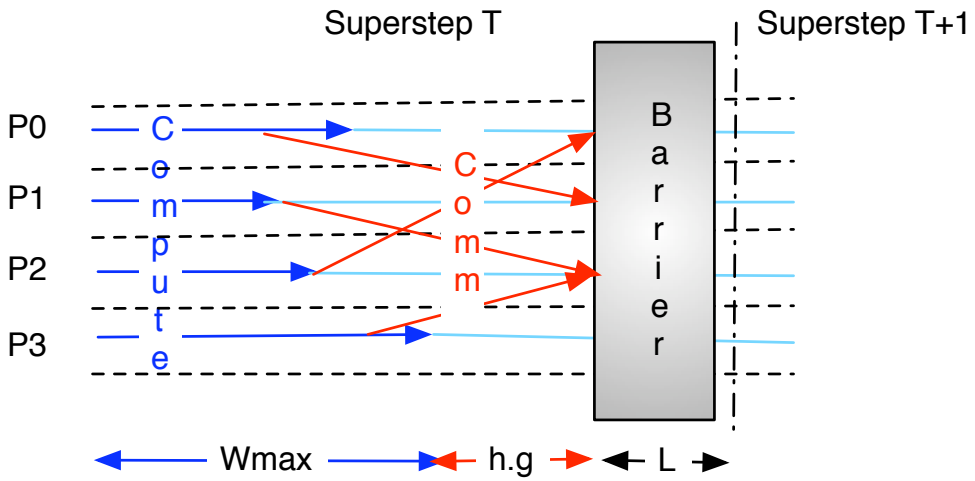

Figure 2.2: Principles of the BSP programming model

- **The machine model** describes a parallel machine as a set of processors linked through a communication medium supporting point-to-point communications and synchronizations. Such machines are described by a set of parameters [Hill 1998]: P (number of processors), r (processor speed), g (communication speed) and L (synchronization duration).

- **The programming model** specifies how a parallel program is structured (Figure 2.2). A BSP program is a sequence of super-steps in which each process performs local computations followed by communications. When all processes reach the synchronization barrier, the next super-step begins.

- **The cost model** provides an analytical way of estimating the runtime cost of a BSP algorithm. Following this model, the cost of a super-step is determined as the sum of the cost of the longest running local computation, the cost of global communication between the processors, and the cost of the barrier synchronisation at the end of the super-step. The cost of one super-step for p processors is expressed as

$$max_{i=1}^{p}(w_i) + max_{i=1}^{p}(h_i g) + l$$

  where w_i is the cost for the local computation in process i, and h_i is the number of messages sent or received by process i.

As for P-RAM and LOG-P, various extensions of BSP have been proposed by either increasing the precision of the cost model [Blelloch 1997], by reducing the impact of the synchronization step which is the main limitation of the BSP model [Gonzalez 2000] or extending it to heterogeneous systems [Li 2012]. Various implementations of BSP have been proposed. Table 2.1 gathers information about those implementations.

Implementation	Model	Abstraction Level	Target Language	Architecture
BSPlib[Hill 1998]	BSP	Low	C, C++ FORTRAN	Cluster
BSPonMPI[Suilen 2006]	BSP	Low	C, C++	Cluster
BSPK[Fahmy 1996]	Oblivious BSP [Gonzalez 2000]	Low	C, C++	Cluster
BSPGreen[Goudreau 1999]	BSP	Low	C, C++	Cluster
PUB-BSP[Bonorden 1999]	BSP	Medium	C, C++	Cluster Embedded Systems
BSML[Loulergue 2002]	BSP	High	OCaml	Cluster

Table 2.1: Available Implementations of the BSP model

2.3 Memory Centric Abstraction

A second class of parallel programming models uses the properties of memory hierarchies and the distribution properties to guide developers in the design of parallel programs. They are often implemented as new languages or existing language extensions due to their interaction with low level memory management or networking systems.

2.3.1 HTA

Hierarchically Tiled Arrays or HTA is a parallel data type designed to facilitate the writing of programs based on tiles in object-oriented languages [Bikshandi 2006]. HTA allows to exploit locality as well as to express parallelism with much less effort than other approaches. Implementations of the HTA model have been developed for C++ and MATLAB, with the parallel back-end running on MPI. A C++ version for shared-memory systems based on Intel Threading Building Blocks has been developed. A Hierarchically Tiled Array or HTA is an array that is subdivided into tiles. Each tile can be either a regular array or another HTA, hence the recursive nature of this data type [Brodman 2008]. By leading developers to handle arrays as an explicit or implicit aggregation of tiles, it simplifies the exploitation of locality hints in parallel programs.

Listing 2.1 illustrates a Jacobi computation with HTAs. A and B are HTAs with one level of tiling; there are n tiles at the root of the tiling hierarchy (level 0), each tile holding $d+2$ variables (level 1). Variables at index 0 and $d+1$ in each tile are

ghost cells. The boundary exchange first updates the ghost cells at index 0, then at index $d+1$. The iteration across tiles is implicit in all assignments.

Listing 2.1: Jacobi update C++ implementation using HTA

```
HTA<double,1> A(, B;
double S = 0.125;

while (!converged)
{
  Tuple<1> t1n(1,n), t0n1(0,n-1);
  Tuple<1> t1d(1,d), t0d1(0,d-1), t0d1(2,d+1);

  // boundary exchange
  B(t1n)[0]    = B(t0n1)[d];
  B(t0n1)[d+1] = B(t1n)[1];

  // stencil computation
  A()[t1d] = S * (B()[t2d1] + B()[t0d1]);

  // ...
}
```

In the stencil computation, the region is not specified at tile access and thus all tiles at level 0 are considered in the operation.

2.3.2 PGAS Languages

Partitioned Global Address Space (PGAS) languages [Blagojević 2010] are designed around a memory model in which a global address space is logically partitioned such that a portion of it is local to each processor. Those languages rely on the use of one-sided communications provided by various distributed memory run-times. PGAS abstracts away the notion of communications by providing primitives to build distributed data structures but still requires a SPMD programing style that may limit its applicability. Three PGAS based languages are usually put forward:

2.3.2.1 Co-Array Fortran

Co-Array Fortran [Numrich 1998] supports the ability to refer to the multiple co-operating instances of an SPMD program (known as images) through a new type of array dimension called a co-array. By declaring a variable with such dimension, the user can specify how each image will allocate a copy of the variable. Remote accesses between variables are performed by indexing over this co-array dimension using square brackets instead of the classical parens of Fortran. In addition to this, synchronization primitives are provided to coordinate images.

2.3.2.2 UPC

UPC is a C extension for supporting PGAS-style computation [El-Ghazawi 2003]. Contrary to Co-Array Fortran, UPC relies on automatic distribution of specially typed instances of array following a linear, block cyclic manner. If this makes distribution of works simpler, UPC distributed array suffers from classical C limitations

and a task like 2D distribution of a 2D array is tedious. This can be simplified by using UPC PGAS pointer that can point to any shared or global address space while maintaining locality information in the pointer type. Finally, UPC also supports a new loop structure – `forall` – which allows the distribution of work to workers by following affinity rules.

2.3.2.3 Titanium

Titanium is a PGAS dialect for Java [Yelick 1998]. Titanium adds several features to Java to support multidimensional arrays, iterators, sub-arrays, copying, operator overloading and performance-oriented memory management. The distributions of data and synchronization between SPMD program instances are provided through similar type marking as UPC, allowing the JAVA compiler to statically enforce synchronization.

2.4 Pattern Centric Abstraction

Patterns based programming models are based on the observation that parallelism is expressed in the form of a few recurring patterns of computation and communication found in a large number of applications. Different kinds of patterns can be extracted and proposed as composable entities. Pattern abstractions also have the interesting properties to be defined on top of other kind of parallel programming models, *e.g.* pattern rising from BSP algorithms.

2.4.1 Parallel Skeletons

The concept of Parallel Skeletons [Cole 1989] is based on the very definition of Pattern Centric Abstractions. In this model, every application domain has its own specific skeletons – or patterns – that can be leveraged and combined. For example, in computer vision, parallel skeletons mostly involve slicing and distributing regular data while parallel exploration of tree-like structures is common in operational research applications. The main advantage of this parametrization of parallelism is that all low-level, architecture or framework dependent code is hidden from the user, who only has to write sequential code fragments and instantiate skeletons.

Another interesting feature of skeletons is their ability to be nested. If we look at a skeleton as a function taking functions as arguments and producing parallel code, then any instantiated skeleton is eligible as being another skeleton's argument. Skeletons are thus seen as *higher-order functions* in the sense of functional programming. At the user's level, building parallel softwares using algorithmic skeletons boils down to simply combining skeletons and sequential code fragments. Classical skeletons includes:

- The `map` skeleton encapsulates classical, SIMD style data parallelism. `map` applies, in parallel, a function f to each element of an array or list-like data

structure. Parallelism is then exploited as the application of f on two distinct elements is done simultaneously. In computer vision, a variant of this skeleton called `scm` is often found and operates on the whole image, performing the slicing and merging of the input and output image.

- The `pipeline` skeleton models situations in which a sequence of successive tasks are dispatched and run in parallel on a set of processing units. Each task receives its input from its predecessor and sends its result to the next processor in the pipeline, either in a blocking or non-blocking way.

- The `farm` skeleton [Poldner 2005] handles load balancing in data-parallel context, *i.e.* situations in which a –potentially ordered– list of items has to be processed by a pool of workers following some kind of load-balancing strategy. Each item is dispatched to the first non-busy `worker` and a result is sent to an implicit collector each time a `worker` task is completed.

- The `par` skeleton is a generic *ad-hoc* skeleton for running N different tasks on a subset of N processing units. No implicit communication is provided and all synchronization or data transfer should be explicitly carried out by the inner tasks. This skeleton is a way to integrate *ad-hoc* parallelism [Cole 2004] in skeleton-based applications.

As an example of Skeleton efficiency at capturing parallel applications structure in a compact way, consider the process graph on figure 2.3.

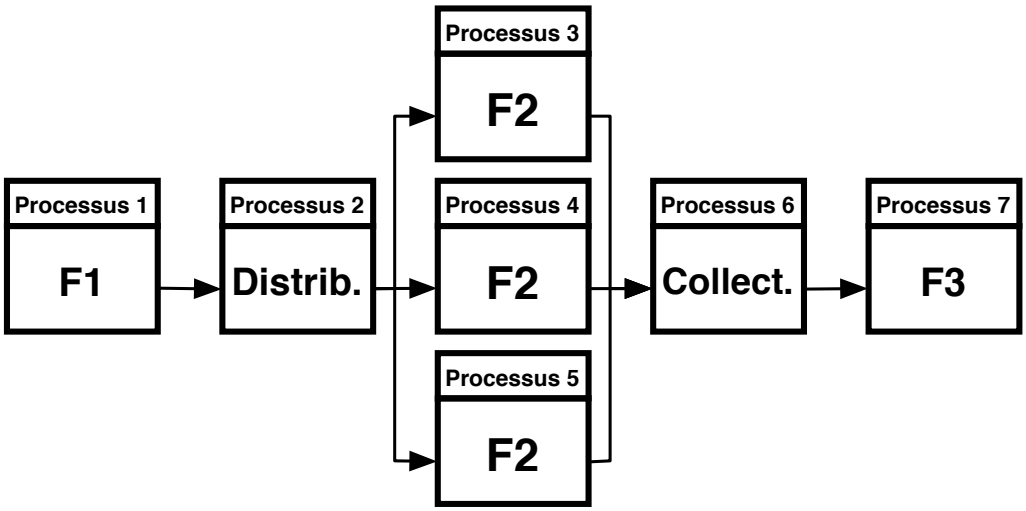

Figure 2.3: A sample process graph

This graph describes an application in which a stream of data are processed by process 1, passed to a distribution process that will send the data to one of the three slave process, gather the results then pass them to the final process. The first level

of communications between process 1, 2,6 and 7 is no less than a `pipeline`. The group of process 2 to 6 can be seen as a `farm` made of three slaves (fig. 2.4).

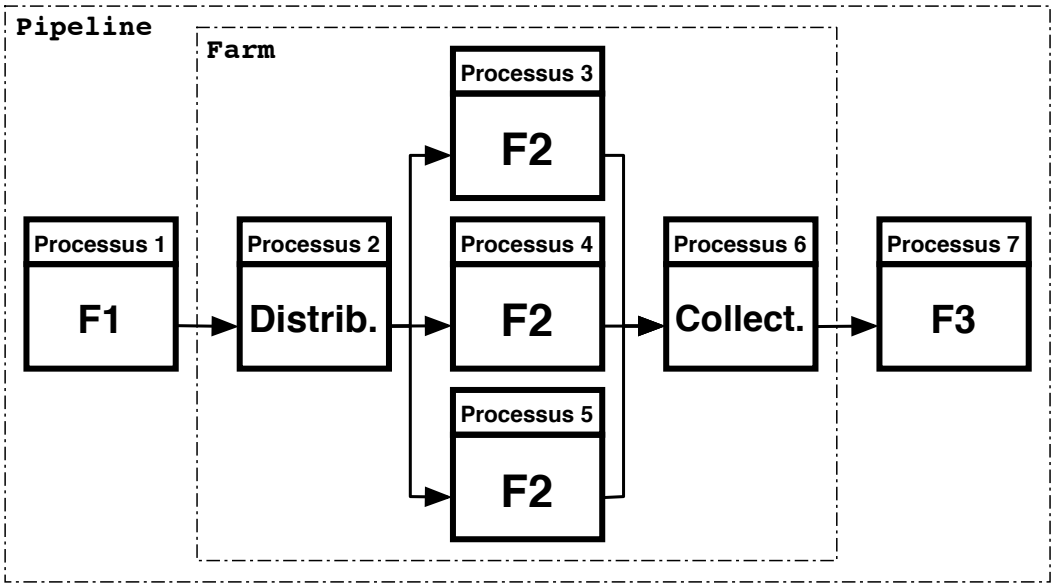

Figure 2.4: A sample process graph viewed as nested skeletons

This sample application can then be expressed as a language agnostic, skeleton-based description :

$$pipeline(f_1, farm(3, f_2), f_3)$$

This terse description, backed up by the skeletons semantic, describes the whole application structure and behavior. Note that the distribution and collection process don't appear as they are mere artifact of one potential farm implementation.

A large body of works around Skeletons exists and covers a large selection of domains and languages. As an exhaustive study of the properties of skeleton based tools has been performed in [Gonzalez-Velez 2010], we present here the most influential tools.

- The **Edinburgh Skeleton Library** [Cole 2004, Benoit 2005b] (eSkel) is a C library based on MPI that provides a set of skeletons including: `pipeline`, `farm`, `deal`, `butterfly` and `haloSwap`. It introduced a large number of fundamental skeleton related notions like different types of nesting [Benoit 2005a] (transient or persistent) and attempted at providing a cost model based on process algebra for its skeletons [Benoit 2008].

- The **Pisa Parallel Programming Language** [Bacci 1995] (P3L) provides skeleton constructs that are used to coordinate the parallel or sequential execution of C code. A compiler named Anacleto [Ciarpaglini 1997] is provided and uses implementation templates to compile P3L code on target architectures.

A performance model can then be used to decide program transformations for optimizations. Users can define a P3L module as a properly defined skeleton construct with input and output streams, and other sub-modules or sequential C code. Modules can be nested using the two tier model, where the outer level is composed of task parallel skeletons, while data parallel skeletons may be used in the inner level.

- The **Muenster Skeleton Library** [Ciechanowicz 2009] (Muesli) is a C++ template library supporting higher order functions, currying, and polymorphic types and uses both MPi and OpenMP to implement both task and data parallel skeletons, using a nesting approach similar to the two tier approach of P3L. In Muesli, C++ templates are used to render skeletons polymorphic, but no type system is enforced. The supported skeletons include Branch & Bound, Divide & Conquer, Farm, Pipe, Filter, reduce, map, permute, zip and their variants. Currently, Muesli supports distributed data structures for arrays, matrices, and sparse matrices.

- **Sketo** [Matsuzaki 2006] is a C++ skeleton library focusing on parallel data structures as such as: lists, trees, and matrices [Emoto 2007]. The data structures are generic, and operations like map, reduce, scan, zip, shift can be applied in parallel on their contents. SkeTo also uses C++ genericity to implement optimization rules like the skeleton fusion transformation [Matsuzaki 2004] , which merges two successive function invocations into a single one, thus decreasing the function call overheads and avoiding the creation of intermediate data structures passed between functions.

- **Marrow**[Marques 2013] is a C++ parallel skeleton library for heterogeneous, multi-GPU environments relying on OpenCL. Marrow provides a set of classical data and task parallel skeletons including `map` and `pipeline` and supports nesting. Marrow automatically generates all the host/device communication and orchestration code, including communications overlap and code generation.

2.4.2 Futures and Promises

Experience with parallel programming has shown that common synchronization techniques like barriers do not scale well on massively parallel machines [Beckman 2006], with thousands of workers. One would like to use finer grain synchronization, but reasoning about the exact point an operation will complete is virtually impossible in a parallel environment of large scale. An alternative is to use asynchronous programming models, which allow the programmer to write programs where a thread or process can be oblivious to what actions the other threads/processes are doing. However, he should still be able to retrieve the results of concurrent works and produce the correct result.

The **Futures and Promises** model is such an asynchronous programming model. A `Future` is a special variable which may or may not have a value at the time that it is referenced in program. A promise is a special construct that is associated with a `Future` and can be used by another thread or process to set the value of the `Future` variable. Usually, the `Future` is used only to read the variable value, while the promise is used to write to the same variable, thus defining a data-flow relation between different threads/processes. The promise construct is often hidden from the programmer. Instead he will have to declare a callable object (function, functor, etc). The library will offer a mechanism to use this callable object to set the `Future` through the promise, after executing the user's callable object. Such is the use of the *async* function in the C++ 11 standard, where the user can issue a function or functor object and retrieve a `Future` object using the *async* call. The *async* will be run by a thread, and the return value of the function or functor will be used to set the `Future` object associated with that*async* call.

An important design decision for any `Futures` implementation is what happens when a `Future` is referenced, while its value is not yet available. A common choice is to have the caller block until the `Future` value is resolved or implicitly try to resolve the `Future` at the reference time (as with lazy evaluation schemes). Figure 2.5 shows the execution model of the blocking scheme.

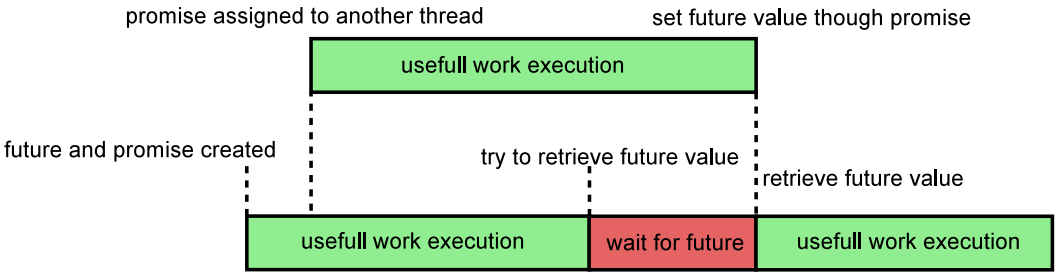

Figure 2.5: The `Futures` execution model of the blocking schematics.

The green color is the time a thread spends doing useful computation, while the red color is the idle time a thread spends on waiting for the result of the `Future`. This is the scheme used by C++ 11 and in the Scala `Future` implementation [Philipp 2012], where the user can set a callable object to be called when the `Future` will be set, or if the `Future` throws an exception (failure), using the callback mechanism. This scheme has the benefit that there will be no blocking at any point of the code, allowing true asynchronous execution. The C++ 11 standard, as most blocking `Future` implementations, offers the option to ask whether a `Future` is ready before referencing its value, in order to avoid any blocking if possible.

Other than their asynchronous execution model, `Futures` offer an easily programmable and expressive user interface that is very close to sequential programming. As a motivation to the reader, we present in Listing 2.2 a Fibonacci func-

tion implementation, using the C++ 11 standard threads library [Committee 2011] `Future` interface.

Listing 2.2: A fibonacci implementation using the C++ 11 futures interface

```
int fibonacci(int n)
{
  if(n == 0) return 0;
  if(n == 1) return 1;

  std::future<int> fib1 = std::async(fibonacci, n-1);
  std::future<int> fib2 = std::async(fibonacci, n-2);

  return fib1.get() + fib2.get();
}
```

Listing 2.3 also shows the sequential equivalent for comparison.

Listing 2.3: Sequential Fibonacci

```
int fibonacci(int n)
{
  if(n == 0) return 0;
  if(n == 1) return 1;

  return fibonacci(n-1) + fibonacci(n-2);
}
```

The parallel version simply requires the recursive calls to be issued using the *async* function, and the use of the get method on the `Future` objects in order to retrieve the return values of the recursive calls. Note that the call to the get method here is blocking.

2.5 Conclusion

Design and exploration of parallel programming models have been a very productive research area for more than thirty years. Various approaches targeted at solving different issues have been proposed and implemented. With respect to our initial goal, the following models appear as strong contender as the basis of new tools' development:

- the BSP model for its ability to provide an analytical cost model that can be used to drive optimizations;
- parallel skeletons which abstraction level and composability could help handling the hierarchical and heterogeneous nature of modern parallel hardware;
- `Futures` which provide a sound and composable programming model for asynchronous operations

The following chapters will investigate how such models can cooperate and be used as the underlying layer of new tools and how modern development practices can accommodate those models while delivering high performances.

Chapter 3

Modern C++ Design Techniques

Contents

" Within C++, there is a much smaller and cleaner language struggling to get out."

— Bjarne Stroustrup, *The Design and Evolution of C++.*

3.1 Objectives

The previous chapter introduced different parallel programming models, from which we selected a subset amenable to proper implementation. We also noticed that a large number of C++ based implementations of library or tools using those models are available. Some of them use C++ features like generic programming or template meta-programming to achieve high levels of performances or to define intuitive API. This chapter will go over the definition of such modern C++ design techniques and their respective advantages.

3.2 Generic Programming

Generic Programming is a programming paradigm for developing efficient, reusable software libraries. Pioneered by Alexander Stepanov and David Musser, Generic Programming obtained its first major success when the Standard Template Library became part of the C++ Standard [Stepanov 1995a]. This process focuses on

finding commonality among similar implementations of the same algorithm, then providing suitable abstractions so that a single, generic algorithm can cover many concrete implementations. This process, called lifting, is repeated until the generic algorithm has reached a suitable level of abstraction, where it provides maximal re-usability while still yielding efficient, concrete implementations. The abstractions themselves are expressed as requirements on the parameters to the generic algorithm.

This notion of requirements are then promoted to first class entities called Concepts [Gregor 2006] that carry the semantic informations about the types requirements, thus providing more than just a generic, reusable implementation, but a better understanding of the problem domain. By building a library API and algorithms on Concepts usually lowers the burden of implementation by requiring the strict minimal set of interfaces from the user defined types, is efficient as runtime checks of interface –as done via dynamic polymorphism– are replaced by simple syntactic verifications and, finally, is far more extensible as new types can be adapted to fit any given Concepts without requiring heavy inheritance.

As an example, consider writing a C++ function that computes the sum of all the elements contained between two memory location (for example, the beginning and the end of an array). To do so in a generic way, we rely on the C++ Concepts of `InputIterator`. With generic programming, the concrete interface of an `InputIterator` is irrelevant. The code is written using templates accepting any types and is properly instantiated as long as those types model the correct set of Concepts, regardless of where, when and how they're defined, and whether or not they derive from a common base class or interface.

Iterators in general, and `InputIterator` in particular, can be thought of as an abstraction of pointers. A type `It` is said to **satisfy** the `Inputterator` Concept if it fulfills the following constraints :

- The type `It` satisfies `CopyConstructible` (*i.e.* provides a copy constructor)
- The type `It` satisfies `CopyAssignable` (*i.e.* provides a assignment operator)
- The type `It` satisfies `Destructible` (*i.e.* provides a destructor)
- The type `It` satisfies `EqualityComparable` (*i.e.* provides `operator==`)
- Given `i` and `j`, values of type `It&`, the following expressions are valid:
  - `std::iterator_traits<It>::reference r = *i;`
  - `It& j = ++i;`
  - `(void)i++;`
  - `bool c = i != j;`

Compare this with a traditional object-oriented programming, where iterators would have be design as inheriting from some `Inputterator` interface which would prevent us from using raw pointers as iterators, thus limiting the applicability – and performance – of the function. Listing 3.1 gives a possible generic implementation of our sum function using the `Iterator` Concept.

Listing 3.1: C++ generic implementation of sum

```
template<typename InputIterator>
typename std::iterator_traits<InputIterator>::value_type
sum(Inputterator b, Inputterator e)
{
  typename std::iterator_traits<InputIterator>::value_type r;

  while(b != e)
    r += *b++;

  return r;
}
```

Calls of this function can then be performed using raw pointers, iterators extracted from a standard container or any type satisfying the `InputIterator` Concept, like for example, a user-defined type with the proper interface but which dereferencing operator returns a monotonous series of values instead of extracting data from memory.

Generic Programming and Concepts also provide support for Concept Checking, a practice that allows for a limited support of parametrized types constraints [Siek 2005], simplifying error checking at compile-time. Tools like the Boost Concept Check Library or ConceptGCC compiler implement such mechanisms. Latest developments on Concepts by Sutton et al. [Sutton 2011] brought up a revised Concepts implementation and applied it on the C++ standard library, simplifying the work needed to support Concepts in mainstream C++ compiler. Discussions are still ongoing to decide if and how Concepts will be integrated in the upcoming C++ 17 standard.

Generic Programming has been successfully applied to the design of software tools like:

- The **Adobe Generic Image Library**: GIL [Parent 2014] is a C++ generic library that allows for writing generic imaging algorithms with performance comparable to hand-writing for a particular image type. The library is designed to be flexible and efficient by using abstract image representations from algorithms on images. It allows for writing code once and having it work for any image type. These image types can then be specified through compile-time or run-time options and policies that helps the library to select and optimize the proper code. Compatibility with standard C++ library components are also provided, thus allowing GIL to be seamlessly integrated into existing code.

- The **Boost Graph Library**: BGL [Siek 2002] aims at applying the concept of Generic Programming to the design of graph oriented algorithms able to operate on a large selection of graph representations (linked list of nodes, adjacency matrix, arrays, etc ...) . By decoupling the algorithm themselves from the internal representation of the graphs, BGL is able to apply a large selection of graphs based algorithms on various, customizable, graph entity using a large variety of graph representations.

- The **Standard Template Adaptive Parallel Library** (STAPL) is a framework for developing parallel programs in C++ [Buss 2010]. It is designed to work on both shared and distributed memory parallel computers. Its core is a library of ISO Standard C++ components with interfaces similar to the sequential ISO C++ standard library. STAPL includes a run-time system, design rules for extending the provided library code, and optimization tools. STAPL relies on a tiered structure in which parallel containers, parallel algorithms and a supporting run-time system are isolated and allow various levels of users –ranging from standard user to domain expert– to take advantage of large scale parallelism even with complex, non-contiguous data structures.

Note that if properties of values, containers and operations handling those are the most usual Concepts laid by those libraries, few defines Concepts for parallel elements even if these library performs parallel computation.

3.3 Active libraries

In opposition to classic libraries, *Active libraries* [Veldhuizen 1998] takes an active role during the compilation phase to generate code. They aim at solving the abstraction/efficiency trade-off problem. They base their approach on defining a set of generative programming methods. These libraries provide domain-specific abstractions through generic components and also define the domain-driven generator to control how these components are optimized. By carrying domain-specific semantic at a high level, this technique enables a semantic analysis of the code before any real code generation process kicks in. Such informations and transformations are then carried on by a meta-language that allows the developer to embed meta-informations. Once the generator finds a solution space in the configuration space, the code generation phase starts resulting on an optimized version of the code. The main approach to design such libraries is to implement them as Domain Specific Embedded Languages (*DSEL*s). As they reuse general purpose language features and existing compilers, *DSEL*s are easier to design and implement.

3.3.1 Domain Specific Embedded Languages

By definition, a Domain-Specific Language (*DSL*) is a computer language specialized to a particular application domain, contrary to a general-purpose language, which is broadly applicable across domains, and lacks specialized features for a particular domain. Domain Specific Embedded Languages (*DSEL*s) are a subclass of *DSL* that rely on an existing general-purpose language to host it. *DSEL*s then reuse the host language syntax and tool ecosystem to be compiled or interpreted. The compile-time process of generating new code (either inside or outside the current host language) known as **Template Meta-Programming** is then used to ensure performances and correctness.

3.3.2 Template Meta-Programming

C++ template meta-programming [Abrahams 2004] is a technique based on the abuse of the template type system of C++ to perform arbitrary computation at compile time. This properties of template is due to the fact that C++ templates define a Turing-complete sub-language manipulating types and integral constants at compile-time [Unruh 1994]. Due to the fact that template code generation is performed at compile-time, uses constants and supports pattern-matching and recursion thanks to template partial specialization, template can also looked at as a pure functional language [Haeri 2012].

Templates are an interesting technique for generative programming. As templates are Turing-complete, one can design a set of templates meta-programs acting as a *DSL* compiler run at compile-time and generating temporary C++ code fragment as an output. The resulting temporary source code is then merged with the rest of the source code and finally processed by the classic compilation process. Through this technique, compile-time constants, data structures and complete functions can be manipulated. The execution of meta-programs by the compiler enables the library to implement domain-specific optimizations that lead to a complete domain oriented code generation. Such a technique can be hosted by several languages featuring meta-programming features (incidental or by design) like D [Bright 2014], Haskell [Sheard 2002] and OCaml [Serot 2008].

*DSEL*s in C++ use template meta-programming via the *Expression Template* idiom. **Expression Templates** [Veldhuizen 1995, Vandevoorde 2002] is a technique implementing a form of **delayed evaluation** in C++ [Spinellis 2001]. Expression Templates are built around the *recursive type composition* idiom [Jarvi 1998] that allows the construction, at compile-time, of a type representing the abstract syntax tree of an arbitrary statement. This is done by overloading functions and operators on those types so they return a lightweight object which type represents the current operation in the Abstract Syntax Tree (AST) being built instead of performing any kind of computation. Once reconstructed, this AST can be transformed into

arbitrary code fragments using Template Meta-Programs (see figure 3.1).

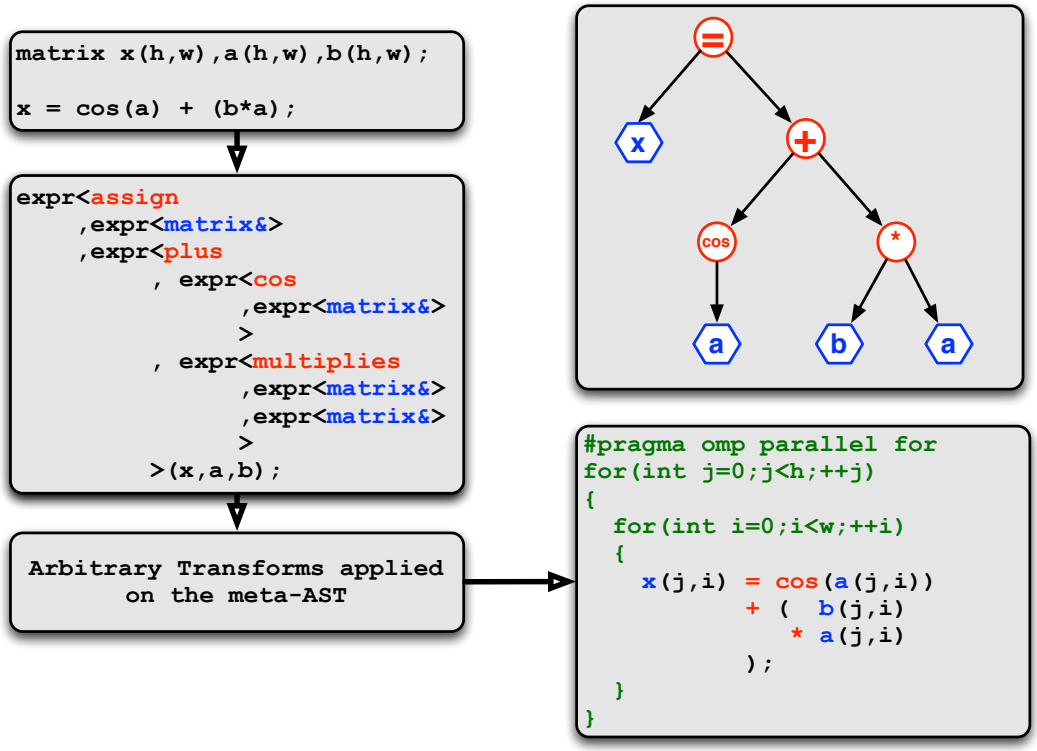

Figure 3.1: General principles of *Expression Templates*

3.3.3 Boost.Proto

While Expression Templates should not be limited to the sole purpose of removing temporaries and memory allocations from C++ code, few projects actually go further. The complexity of the boilerplate code is usually as big as the actual library code, making such tools hard to maintain and extend. To avoid such a scenario, tools encapsulate the Expression Template technique as reusable frameworks with extended features.

The Portable Expression Template Engine or PETE [Haney 1999] extends the expression template technique and provides an engine to handle user defined types in expression statements. It is used in the POOMA framework [Reynders 1996] that provides a set of C++ classes for writing parallel PDE solvers. With PETE, the user can use the engine and apply transformations at the AST level. PETE presents some limitations and its engine does not allow the user to perform common transformations on the AST as it only evaluates expressions with a bottom-up approach. This engine also lacks of domain specific consideration while manipulating expressions.

To alleviate these shortcomings, Niebler has proposed a C++ compiler construction toolkit for embedded languages called Boost.Proto [Niebler 2007]. It allows developers to specify grammars and semantic actions for *DSEL*s and provides a semi-automatic generation of all the template structures needed to perform the AST capture. Simply put, Boost.Proto can be seen as a *DSEL* to design *DSEL*s . Compared to hand-written Expressions Templates-based *DSEL*s , designing a new embedded language with Boost.Proto is done at a higher level of abstraction by designing and applying **Transforms** that are functions operating via pattern matching on *DSEL* statements. In a way, Boost.Proto supersedes the normal compiler workflow so that domain-specific code transformations can take place as soon as possible. The main idea behind Boost.Proto is the construction of an AST structure through the use of terminals. A Boost.Proto `terminal` represents a leaf of an AST. The use of a `terminal` in an expression infects the expression and builds a larger Boost.Proto expression. These expressions are tied to specific domains as Boost.Proto aims at defining *DSEL*s . To illustrate the possibilities of the library, we present a simple analytical function *DSEL* written with Boost.Proto . This *DSEL* will allow us to evaluate analytical expressions of the following form:

```
(x*5 + 2.0*x - 5)
```

We will specify the value of `x` by using the parenthesis operator and it will also triggered the evaluation of the expression like in the following example:

```
(x*5 + 2.0*x - 5)(3.0)
```

Boost.Proto can be seen as a compiler in the sense that it provides a similar way to specify your own language. In comparison to classic compilers, the first entry point of the library is the specification of `grammar` rules. Boost.Proto automatically overloads all the operators for the user but some of them may not be relevant for a DSL. This means that it may be possible to create invalid domain expressions. Boost.Proto can detect invalid expressions through the use of a Boost.Proto `grammar`. A `grammar` is defined as a series of valid grammar elements. In our example, we want to allow the use of:

- classical arithmetic operators;
- analytical variables;
- numeric literals.

We then define a `grammar` that matches these requirements: it is presented in listing 3.2.

Listing 3.2: Analytical grammar with Boost.Proto

```
1  // Terminal type discriminator
2  struct variable_tag {};
3
4  struct analytical_function
5    : boost::proto::or_
6    <
7      boost::proto::terminal< variable_tag >
8
9    , boost::proto::or_
10       < boost::proto::terminal< int >
11       , boost::proto::terminal< float >
12       , boost::proto::terminal< double >
13       >
14
15   , boost::proto::plus<analytical_function,analytical_function>
16   , boost::proto::negate<analytical_function>
17   , boost::proto::minus<analytical_function,analytical_function>
18   , boost::proto::multiplies<analytical_function,analytical_function>
19   , boost::proto::divides<analytical_function,analytical_function>
20 >
21 {};
```

At line 7 of listing 3.2, we allow all terminals that hold a `variable_tag`. This type enables the discrimination between analytical variables and other terminals. At line 9, we allow numeric literals in our expressions. For this specific case, Boost.Proto wraps the literals in terminals. We finally allow the arithmetic operators.

Boost.Proto can now construct valid ASTs. These expression trees do not encapsulate any domain semantic for now. The AST type is a raw tree as if it was extracted from the work-flow of a compiler. The library allow us to add domain semantic to an AST through the declaration of a user-defined domain and a user-defined expression class. This process allows the user to merge the domain-semantic information with the raw structure of an expression. The next step consists in specifying the domain for our analytical DSL. This is done by inheriting from `proto::domain` and linking this domain to an expression generator of a user defined expression type. Listing 3.3 shows the domain declaration.

Listing 3.3: Domain definition with Boost.Proto

```
1  template<typename AST> struct analytical_expression;
2
3  struct  analytical_domain
4        : boost::proto::domain< boost::proto::generator<analytical_expression>
5                              , analytical_function
6                              >
7  {};
```

Once the domain declaration is done, we can now build our `analytical_expression` class. We add a specific interface to this class as we want to be able to call the `operator()` on an expression to evaluate it with a given set of variables. It does not provide the definition of the `operator()`: we will

see how we evaluate our expression later. At this point, we do not provide any particular behavior to this operator. Listing 3.4 presents the `analytical_expression` class that inherits from `proto::extends`. `proto::extends` is an expression wrapper that imbues an expression with analytical domain properties.

Listing 3.4: User-defined expression type with Boost.Proto

```
template<typename AST>
struct  analytical_expression
      : boost::proto::extends< AST
                             , analytical_expression<AST>
                             , analytical_domain
                             >
{
  typedef boost::proto::
          extends< AST
                 , analytical_expression<AST>
                 , analytical_domain
                 > extendee;

  typedef double result_type;

  analytical_expression(AST const& ast = AST()) : extendee(ast) {}
  BOOST_PROTO_EXTENDS_USING_ASSIGN(analytical_expression)

  typedef double result_type;
  result_type operator()(double v0) const;
};
```

Now, we need to implement `operator()` so that Boost.Proto can evaluate the value of our analytical expressions. Boost.Proto handles that by providing **Transforms** that specify rules that need to be performed when the AST is evaluated. A Transform is a *Callable Object* [Standard 2014] defined in the same way that a Boost.Proto grammar. Transform rules can be extended with a semantic action that will describe what happens when a given rule is matched. The library provides a lot of default transforms that we will use in our example. Our transform that evaluates our expression needs to behave differently while walking the AST and encountering its nodes:

- If it is a terminal, we want to extract the corresponding value;
- If it's an operator, we want it to do what the C++ operators does.

To achieve this, we write the `evaluate_` transform that relies on the use of default transforms. `proto::when` is used here to associate a rule to a specific action. The `evaluate_` transform is presented in listing 3.5.

Listing 3.5: The `evaluate_` transform

```
1  struct  evaluate_
2        : boost::proto::or_
3          <
4            boost::proto::when
5            < boost::proto::terminal< variable_tag >
6            , boost::proto::_state
7            >
8          , boost::proto::when
9            < boost::proto::terminal< boost::proto::_ >
10           , boost::proto::_value
11           >
12         , boost::proto::otherwise< boost::proto::_default<evaluate_> >
13         >
14 {};
```

If we want to evaluate an expression like `(x+2.0*x)(3.0)`, we need to evaluate each node and accumulate the result while we walk the AST. Transforms related to accumulation are common when processing ASTs. Boost.Proto provides a clear way to achieve these transforms: the `_state` of an AST. In our case, the `_state` is used at line 6 to pass the value of the analytical variable through each node and ask each node to evaluate themselves with it (see listing 3.6).

Listing 3.6: `operator()` implementation using `evaluate_`

```
1 result_type operator()(double v0) const
2 {
3   evaluate_ callee;
4   return callee(*this,v0);
5 }
```

The evaluation of the analytical expression `(x + 2.0*x)(3.0)` is performed in the following way.

First, the '+' node is evaluated: `( x(3.0) + (2.0*x)(3.0) )()`.
Then, the '*' node: `( x(3.0) + (2.0*x(3.0)) )()`
And finally, the terminals evaluation is performed: `3.0 + (2.0*3.0) = 9.0`.

We notice the use of `proto::_` (line 9) that permits to match any other terminals that are not analytical variables. In this particular case, we directly extract the value of the terminal. Literals will match such a case. At line 12, we simply tell the library to use the default behavior of operators. At the end, we can write analytical expressions that match the correct grammar and evaluate it. This is done by defining terminals and building an expression using them. Listing 3.7 shows how our small analytical *DSL* in action.

Listing 3.7: Analytical expression in action

```
1 analytical_expression< proto::terminal<variable_tag>::type > const _x;
2
3 std::cout << (_x*3 + 9.0*_x)(2) << "\n"; // Output : 24
```

3.4 Other code generation systems

Several similar ideas can be found in other languages.

3.4.1 Delite

Delite [Brown 2011] is a compiler framework and runtime for parallel embedded domain-specific languages from Stanford University PPL. Delite's goal is to enable the rapid construction of high performance, highly productive *DSL*s . Delite provides several facilities like built-in parallel execution patterns, optimizers for parallel code, code generators for Scala, C++ and CUDA and a heterogeneous runtime for executing *DSL*s . Boost.Proto and its use of Template Meta-Programming could be seen as C++ equivalent to Delite in the sense that it is built on similar sub-systems. The main difference is the fact that Delite has access to information like variables name or dependencies across statements that Template Meta-Programming can not access currently.

3.4.2 DESOLA

DESOLA [1] is a linear algebra library developed to explore multi-stage programming as a way to build active libraries [Russell 2011]. The idea is to delay library call executions and generate optimized code at runtime, contrary to *DEMRAL* that may maximize compile-time code generation and optimization. Calls made to the library are used to build a *recipe* for the delayed computation. When the execution is finally forced by the need for a result, the recipe will often represent a complex composition of primitive calls. In order to improve performance over a conventional library, it is important that the generated code should be executed faster than a statically generated counterpart in a conventional library.

3.4.3 TOM

TOM is a language extension designed to manipulate tree structures and XML documents [Moreau 2003]. It provides pattern matching facilities to inspect objects and retrieve values in C, Java, Python, C++ or C#. Tom is a language extension which adds a new matching primitives to C and Java: `%match`. This construct is similar to the match primitive found in functional languages. The patterns are used to discriminate and retrieve information from an algebraic data structure. Therefore, Tom is a good language for programming by pattern matching. This techniques is similar in practice to what Boost.Proto transforms can achieve within C++ itself, thus enabling similar kind of efficient rewrites.

[1] Delayed Evaluation Self Optimizing Linear Algebra

3.5 Conclusion

In this chapter, we introduced various modern C++ design techniques that helps either simplifying API by not requiring a heavy pay-load of virtual inheritance – using Generic programming and Concepts – and to generate efficient code by using AST level introspection of statements and arbitrary code transform from a high-level API to low-level code by using Template meta-programming and *DSEL*s .

ANother important point is also **when** to use such techniques. In the follwoign chapter, we will apply these techniques to the definitions of various high-level parallel programming libraries and we'll try to sketch guidelines on which techniques bring the best abstarction/efficiency ratio.

Chapter 4

The BSP++ Library

Contents

> *"My hypothesis is that we can solve the software crisis in parallel computing, but only if we work from the algorithm down to the hardware – not the traditional hardware first mentality."*
> — Tim Mattson, Intel Principal Engineer

In this chapter, we present the BSP++ library, which is an attempt at applying modern C++ design techniques to the implementation of a parallel programming library based on a high level abstraction. We choose to implement a C++ version of the BSP model as they were no modern implementation using this language and because the BSP abstraction was small enough to allow us to propose various support for different architectures. Work presented in this chapter are the result of Khaled Hamidouche's thesis [Hamidouche 2011a] supervision and its associated papers:

- *"Hybrid bulk synchronous parallelism library for clustered SMP architectures"* [Hamidouche 2010b]
- *"A framework for an automatic hybrid MPI+openMP code generation"* [Hamidouche 2010a]
- *"Three high performance architectures in the parallel APMC boat"* [Hamidouche 2011b]
- *"Parallel biological sequence comparison on heterogeneous high performance computing platforms with BSP++ "* [Hamidouche 2011c]

which contain more detailed implementation description and additional benchmarks.

4.1 The BSP++ Library

4.1.1 Objectives

BSP++ [Hamidouche 2010b] is a lightweight object-oriented implementation of the BSP model that aims to facilitate programming on parallel hybrid architectures. It is based on an extension of BSP for hierarchical and hybrid architectures, close to the `D-BSP` [Beran 1999] model, which can be decomposed in several sub-machines. However, unlike `D-BSP`, BSP++ uses a different value for the parameters of the sub-machines on each level, *i.e.* to take advantage of the hybrid architecture, BSP++ uses different values of L and g on each level of the hierarchy. Furthermore, as many new HPC platforms support collective communication patterns like $all - to - all$, BSP++ takes also advantage of these features to implement collective primitives whenever possible. It supports multi-cores, cluster of multi-cores, CellBE and cluster of CellBE as target platforms.

4.1.2 BSP++ API

BSP++ provides an API to the BSP model based on the notion of **parallel data vector**. In this model, the user stores distributed data in a specialized generic class called `par` that supports the BSP style communication patterns. The BSP++ interface contains the following components:

- `par<T>`: encapsulates the concept of parallel vector. This class can be built from a large selection of C++ constructions ranging from C-style array, C++ standard container, function or lambda-function. The local accesses to a parallel vector data are done through the dereferencing operator;
- `pid_`: a global parallel vector that stores the processor identifier of the current processors;
- `sync`: performs an explicit synchronization, ending the current super-step;
- `proj` returns a function object that maps the identifier of a processor to the contents of the parallel vector held by this processor and ends the current super-step. After a call to `proj`, all processors of a BSP machine have a local copy of the distributed parallel vector values;
- `put`: allows the local values to be transferred to any other processor and ends the current super-step. The return of `put` is a parallel vector of function object of type `T(int)` that returns the data received from processor i when applied to i. Contrary to `proj`, this primitive allows any kind of communication scheme.

The whole BSP semantic is fully encapsulated within these few components and functions, thus limiting the impact of the library on the user code. The interface with both standard C++ components and C style functions provides an easy integration of legacy code. Also, the BSP++ API does not include any reference

to either MPI, OpenMP or CellBE specific elements. The choice of the target architecture is done via a preprocessing symbol passed to the compiler.

Let's look at how BSP++ API can be used and the benefit of its generic implementation by analysing a simple parallel squared norm of a vector using BSP++ (Listing 4.1).

Listing 4.1: BSP++ implementation of squared norm

```
1  #include<bspp/bsppp.hpp>
2
3  int main( int argc, char** argv )
4  {
5    bsp::init( argc, argv );
6
7    BSP_SECTION ()
8    {
9      bsp::par< vector<double> > v ;
10     bsp::par< double > r;
11
12     // super-step (1) : perform local inner product
13     *r = std::inner_product( v->begin(), v->end(), v->begin() , 0.);
14
15     // ... and do a global exchange
16     auto exch = proj(r);
17
18     // super - step (2) : accummulate partial result
19     *r = std::accumulate( exch.begin(), exch.end() );
20     bsp::sync ();
21   }
22
23   bsp::finalize ();
24 }
```

After starting-up the middleware environment on line 5, we start up a BSP code section with the `BSP_SECTION()` macro. This macro takes care of bringing all the required, environment dependent information into scope for the following code fragment. Then, we create an instance of parallel vector for storing the data and the result (lines 9 and 10). The algorithm is then split up into three parts:

- The first super-step performs a local computation of the inner-product of `v` using the C++ standard algorithm (line 13). Once computed, we perform a global exchange of these intermediate results using the `bsp::proj` primitive (line 16). The `bsp::proj` primitive gathers the local data of all the BSP machine processors and sends them to all other processors, returning an object that locally contains the distributed data values. After completion, `bsp::proj` synchronizes the machine and ends the current super-step.
- The second super-step uses this object as a standard range to generically call `std::accumulate` to perform the final reduction (line 19). An explicit synchronization is then done to end the super-step (line 20).
- The BSP program is finally terminated after calling the middleware finalize function (line 23).

An important point is that all objects returned by the various communication primitives are, by design, usable as both functions and array like objects, allowing them to interface naturally with most modern C++ libraries and idioms (like the standard library algorithm), thus enabling a seamless reuse of legacy code. Note also the lack of architecture specific mark-up or functions.

4.1.3 Support for hybrid programming

One common objection to the use of BSP is the cost of global synchronizations that can become dominant for large parallel machines. To reduce this cost, BSP++ takes advantage of the clusters' hybrid architecture by decomposing the global BSP machine on two-level hierarchical BSP sub-machines. In the upper level, a BSP machine is defined among the nodes of the cluster with the message passing mode. In the lower level, a BSP machine is defined among the cores of a node with the shared memory/accelerator mode. For instance, for a multi-core cluster, BSP++ generates code with MPI at the upper level and OpenMP at the lower level. Similarly, in a cluster of CellBEs, BSP++ uses MPI and the Cell SDK in the upper and the lower levels, respectively. In this hybrid mode, the cost model of BSP is modified as follows:

$$\delta = W_{max} + \sum_{i \in Arch} (h_i.g_i(P_i) + L_i(P_i))$$

In this formula, h_i, g_i, L_i and P_i stand respectively for the amount of bytes sent, the communication cost, the synchronization cost and the number of processing elements at the architectural level considered.

The support for hybrid code is done when one uses a BSP sub-machine enabled super-step function inside another BSP sub-machine level super-step computation phase. as shown in Figure 4.1. To enable hybrid computation MPI+OpenMP (Figure 4.1(b)), the computation phase of the MPI super-step is replaced by a call to an OpenMP BSP++ function. In general, nothing more is needed but, in some cases, a split function is used to decompose MPI data into private OpenMP thread variables by passing a range or pointers to the OpenMP step. As the copy time increases with the data size, we split MPI data into each private OpenMP thread variables by passing a ranges or iterators to the OpenMP step, thus minimizing these copies. However, this method can induce false sharing and an overhead induced by the shared access to the data. As this issue is very application dependent, we let the user benchmarks and decides which strategy to apply. For clusters of CellBE, BSP++ also uses a hybrid code with a two-level hierarchy, where the lower level is the kernel executed on the SPEs, and the upper level uses MPI for the communication between PPEs. In this context, the split function between the upper and lower levels uses a `remote_iterator` to split and iterate over the data across the PPE/SPE frontier.

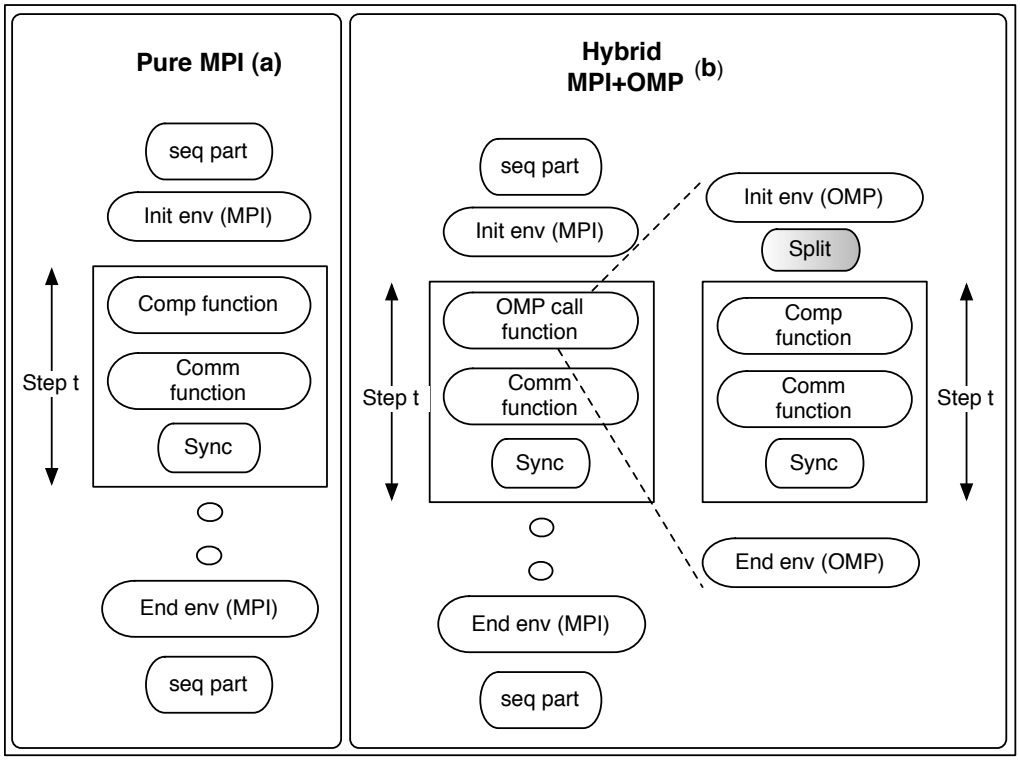

Figure 4.1: Parallelization with the hybrid model

4.2 Benchmarks

We present there two of the most complete application of BSP++ . The first application deals with approximate parallel model checking developed in collaboration with Sylvain Peyronnet. The second application is a bio-informatic application developed in collaboration with Alba Cristina M. A. de Melo from University of Brasilia.

4.2.1 Approximated Model Checking

The goal of probabilistic model checking is to determine the satisfaction probability that a given temporal property holds in a probabilistic system. Since probabilistic systems are very common to model communication protocols, network algorithms, etc. it is of the utmost importance to be able to verify their correctness efficiently. Unfortunately, most probabilistic model checking methods have the drawback of being subject to the so-called state space explosion phenomenon, i.e., the fact that the verification process runs out of memory while verifying large probabilistic systems.

4.2.1.1 Principles of APMC

In this work, we are interested in the quantitative verification problem, *i.e.* , the problem of computing the probability that a probabilistic system, modeled as a Markov chain, satisfies some given linear time temporal formula. One of the first algorithms for this problem was given in [Courcoubetis 1995]. The algorithm transforms step by step the Markov chain and the formula, eliminating one by one the temporal connectives, while preserving the satisfaction probability of the formula. The elimination of temporal connectives is performed by solving a linear system of equations of the size of the Markov chain. Clearly, this algorithm suffers from space complexity issues. And so is the state-of-the-art model checking tool for the verification of quantitative properties `PRISM` [Hinton 2006]. For theoretical and practical reasons, it is then natural to ask the question: can probabilistic verification be efficiently approximated? Approximate Probabilistic Verification for Markov chains has already been investigated in [Hérault 2004] and [Demaille 2006].

For many linear time properties, satisfaction by an execution path of finite length implies satisfaction by any extension of this path. Such properties are called monotone. It is shown that the satisfaction probability of monotone or anti-monotone linear time properties can be approximated with a randomized approximation scheme [Demaille 2006]. Given a Discrete Time or a Continuous Time Markov Chain (DTMC or CTMC) M and a monotone property Ψ, we approximate $Prob[\Psi]$, the probability measure of the set of execution paths satisfying the property Ψ by a fixed point algorithm obtained by iterating a randomized approximation scheme for $Prob_k[\Psi]$, the probability measure associated to the probabilistic space of execution paths of finite length k. We adapt the notion of randomized approximation scheme for counting problems, which is due to Karp and Luby [Karp 1983] to obtain the following random sampling algorithm GAA. It uses the probabilistic generator G for M to compute a good approximation of $Prob_k[\Psi]$ (see table 4.1).

Generic approximation algorithm GAA
Input: G, k, Ψ, ε, δ
Output: ε-approximation of $Prob_k[\Psi]$
N := $ln(2\delta)/2\varepsilon^2$
A := 0
For i = 1 to N do
G.generate_path(σ,k)
If Ψ is true on σ then A := A + 1
Return A/N

Table 4.1: General APMC algorithm

The APMC model checker implements the algorithm described above. Two versions coexist: APMC 3.0 [Hérault 2006] and APMC-CA [Borghi 2008]. The only difference is that the second one has been tuned by hand in order to work

on the Cell architecture. APMC input language is the same as `PRISM` and models under verification are discrete or continuous time Markov chains. Using APMC we compute an approximate value of the probability of a temporal property over a probabilistic system. Basically APMC works by generating random paths in the probabilistic space underlying the system and computes a random variable which estimates the probability of the formula. As stated in [Hérault 2006], we use for that purpose a diagram: a succinct representation of the system. This notion of succinct representation is of utter importance for APMC, since it allows to use only very limited memory. Parallelizing this algorithm mainly consists in generating and verifying paths independently. This is the approach also used by PRISM for the distribution of the simulator [Hinton 2006].

The original APMC software consists of several independent components: the parser, the core library and the deployment tool. The parser is a simple lex/yacc program which parses the PRISM language and LTL formulas. It then calls the core APMC library to produce an internal succinct representation of the model (linear in the size of the model file) and of the properties (linear in the size of the property file). The library then produces the ad-hoc generator and verifier as ANSI C code. The main loop of the code produced by the library consists of generating a path (i.e., a set of configurations) of given length and evaluating the property (linear time formula) on each path. The number of iterations of this loop is a parameter to the program. The output of the ad-hoc program is the number of paths generated, the number of paths where the formula is true, and the number of paths where the formula is false.

4.2.1.2 Performances

The BSP++ implementation is rather simple. Every processing unit will generate a given number of random paths of proper size σ, evaluate the model along this path and aggregate its local value for $\mathbf{A}$. Once done, those local values are gathered using all-to-one projection and averaged. The results of the model checking is then delivered. From there, we expect a rather linear scaling of the implementation, with suboptimal results being explainable by either overheads from the library –which generic programming should have minimized– or properties of the architectures.

The experiments were conducted on two different platforms:

- The first machine, the AMD machine, is a quad-core quad- processor 2GHz AMD Opteron 8354 machine with 4 × 2-MB L3 cache and 16-GB of RAM, running the 2.6.26 Linux kernel and g++ 4.3 with openMP 2.0 support and openMPI. In all our experiments, the task/core mapping was set using the sched affinity system call to prevent thread migration between cores and get stable performance.
- The second machine, the CLUSTER machine, is a cluster of the GRID5000

platform [34]. We used between 2 and 64 nodes connected by a 2 × BCM5704 Gigabit Ethernet Network. Each node is a dual-core bi-processor 2.6-GHz AMD Opteron 2218 with 2 × 2-MB L2 cache and 4-GB of RAM, using the MPICH2.1.0.6 library.

Our experiments consist of the verification of temporal properties on two different models. The first one is the dining philosophers model [Pnueli 1986], for which we check a reachability property. Being very well-known, this model allows us to make sure there are no unattended strange behavior in the verification process. The parameter of this model is the number of philosophers that interact all together. The second model is the sensor network model (see [36]), whose interest is that it is composed of a huge number of small interacting modules. For this second model, we also verified a reachability property. The parameter of this model is the size of the communication grid, meaning that the model SNX contains X sensors. Both models have an explicit representation which is of exponential size with respect to the parameters. The size of the path is taken as a parameter for both models. For the first model the size is 900 and 8000 for the second one.

Figures A.1, A.2, A.3 and A.4 show the slowdown for the MPI version of APMC on the dining philosophers and the sensor network models on the AMD and the CLUSTER machine respectively. On the AMD machine we get a super linear speedup from 1 to 8 cores and a linear speedup to 16 cores. This means that the parallelization scales perfectly: there are no overhead in the use of BSP++ . On the CLUSTER machine, the figures show a linear speedup until 128 cores and a small slowdown from 128 to 256 cores. The slowdown is due to the synchronization time that rises with the number of cores.

The OpenMP version of APMC leads to different slowdowns for both models. These results are depicted in Figures A.5 and A.5. The results show a super linear speedup for the dining philosophers model and a linear one for the sensor network model. Compared to the MPI version, there is no difference between the execution time on 1 to 8 cores because both experiments are conducted on a shared memory architecture. On 16 cores, the OpenMP version gets a small advantage due to the faster synchronization.

As the CLUSTER machine nodes are dual-core bi-processors, we experimented an hybrid version in which each MPI node starts 4 openMP threads. Compared to the pure MPI version, the hybrid MPI/OpenMP version gets better performances, the reduction of synchronization time by using a two levels synchronization being the key point of this enhancement. The speed-up of the hybrid version is ranging from 50% to 10% compared to the MPI version for the dining philosophers model, from 50 philosophers to 800 respectively. The explanation of the slowdown decrease is as follows: when the number of philosophers is small the communication/computation ratio is big (because the number of modules is exactly the number of philosophers),

which gives an advantage for the hybrid version. On the contrary, when the number of philosophers is high, the computation part is prevailing the communication part, which in turn decreases the advantage of using OpenMP for the communication. The results of slowdown plotted in Figures and show that the hybrid version gets a super linear speedup up to 256 cores.

4.2.2 DNA Sequence Alignment

A biological sequence is a molecule of nucleic acids or proteins. It is represented by an ordered list of residues, which are nucleotide bases (for DNA or RNA sequences) or amino acids (for protein sequences). In particular, DNA sequences are treated as strings composed by elements of the alphabet $\sum = \{A, T, G, C\}$. Since two DNA sequences are rarely identical, sequence comparison is in fact a problem of approximate pattern matching [Mount 2004]. To compare two sequences, we need to find the best alignment between them, which is to place one sequence above the other making clear the correspondence between similar characters. In an alignment, spaces can be inserted in arbitrary locations so that the sequences end up with the same size.

Given an alignment between sequences s and t, a score is associated to it as follows. For each two bases in the same column, we associate (a) a punctuation ma, if both characters are identical (*match*); or (b) a penalty mi, if the characters are different (*mismatch*); or (c) a penalty g, if one of the characters is a space (*gap*). The score is the addition of all these values. The maximal score is called the similarity between the sequences. Figure 4.2 presents one possible global alignment between two DNA sequences and its associated score. In this figure, $ma = +1$, $mi = -1$ and $g = -2$.

A	C	T	T	G	T	C	C	G
A	$-$	T	T	G	T	C	A	G
$+1$	-2	$+1$	$+1$	$+1$	$+1$	$+1$	-1	$+1$

$score = 4$

Figure 4.2: Example of alignment and score

4.2.2.1 The Smith-Waterman (SW) Algorithm

The Smith-Waterman algorithm [Smith 1981] is an exact method based on dynamic programming to obtain the best pairwise local alignment in quadratic time and space. It is divided in two phases: create the similarity matrix and obtain the alignment.

- The first phase of the SW algorithm receives as input sequences s and t, with $|s| = m$ and $|t| = n$, where $|s|$ represents the size of sequence s. For sequences

	*	G	A	A	G	C	T	A
*	0	0	0	**0**	0	0	0	0
G	0	1	0	0	**1**	0	0	0
C	0	0	0	0	0	**2**	0	0
T	0	0	0	0	0	0	**3**	1
G	0	1	0	0	1	0	1	2
A	0	0	2	1	0	0	0	2
C	0	0	0	0	0	1	0	0
C	0	0	0	0	0	1	0	0
T	0	0	0	0	0	0	2	0

Figure 4.3: Similarity matrix for sequences s and t

s and t, there are $m+1$ and $n+1$ possible prefixes, including the empty sequence. The notation used to represent the *i-th* character of a sequence *seq* is $seq[i]$ and $seq[1..i]$ is used to represent a prefix with i characters, from the beginning of the sequence.

The similarity matrix is denoted $A_{m+1,n+1}$, where $A_{i,j}$ contains the score between prefixes $s[1..i]$ and $t[1..j]$. At the beginning, the first row and column are filled with zeroes. The remaining elements of A are obtained from equation 4.1.

$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + (\text{if } s[i] = s[j] \text{ then } ma \text{ else } mi) \\ A_{i,j-1} - \text{g} \\ A_{i-1,j} - \text{g} \\ 0 \end{cases} \tag{4.1}$$

In addition, each cell $A_{i,j}$ contains information (arrow) about the cell that was used to produce the value.

- The second phase is designed to obtain the best local alignment. The algorithm starts from the cell that has the highest value in $A_{i,j}$, following the arrows until the value zero is reached. A left arrow in $A_{i,j}$ is the alignment of $s[i]$ with a gap in t. An up arrow represents the alignment of $t[j]$ with a gap in s. Finally, an arrow on the diagonal indicates that $s[i]$ is aligned with $t[j]$. Figure 4.3 presents the similarity matrix to obtain the alignment between two sequences, with $score = 3$.

4.2.2.2 Parallel Smith-Waterman

In the Smith-Waterman algorithm, most of the time is spent calculating matrix A and this is the part that is usually parallelized. The access pattern presented by the matrix calculation is non-uniform and the parallelization strategy that is

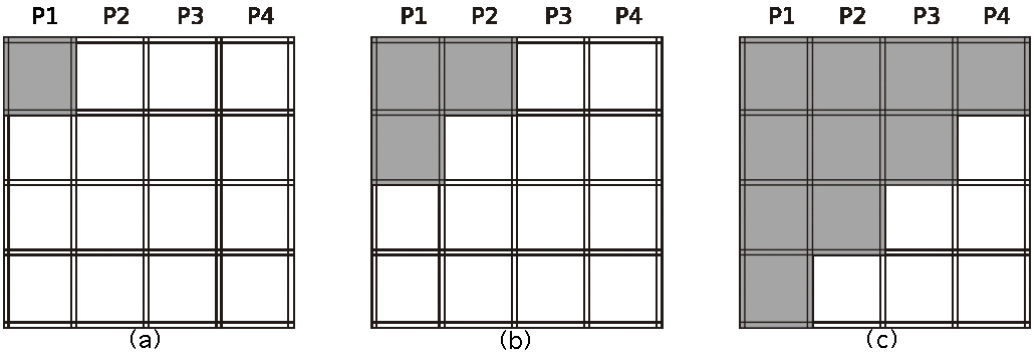

Figure 4.4: The wavefront method.

traditionally used is the wavefront method [Pfister 1995], since the calculations that can be done in parallel evolve as waves on diagonals.

Figure 4.4 illustrates the wavefront method using a column-based block partition technique for four processors. At the beginning, only $P1$ is computing (Figure 4.4.a). When $P1$ finishes calculating the values of a block, it sends its border column to $P2$, that can start calculating (Figure 4.4.b). In Figure 4.4.c, the maximum parallelism is obtained.

4.2.2.3 Performances

We implemented the parallel SW algorithm using BSP++ , generating five different versions: BSP++ MPI, BSP++ OMP, BSP++ MPI+OMP, BSP++ Cell and BSP++ MPI+Cell. The BSP++ MPI version uses MPI primitives for communication among the nodes. The BSP++ OMP version runs on shared memory machines and the communication between the nodes is done exclusively by shared memory. In the BSP++ MPI+OMP version, communication among the nodes is done through MPI and communication among the cores uses OpenMP SPMD style [Hamidouche 2010b]. The experiments were conducted on three different platforms:

- `AMD16`: quad-processor quad-core 2GHz AMD Opteron, 16-GB of RAM and a 3-MB L3 cache running the 2.6.28 Linux kernel and g++ 4.4 with OpenMP 2.0 support and OpenMPI 1.4.2. In all our experiments, the task/core mapping was set using the `sched affinity` system call to prevent thread migration between cores and get stable performance.

- `CLUSTER128`: a 32-node cluster from the Bordeaux site of the GRID5000 platform [Cappello 2010]. Each node is a bi-processor bi-core 2.6-GHz AMD Opteron, 4-GB of RAM and a 2-MB L2 cache using g++-4.4 with OpenMP 2.0 support and the OpenMPI 1.4.3 library.

- `HOPPER`: a 6384-node Cray XE6 cluster, where each node is composed of 24 cores (2 x 12 AMD 2.1-GHz), with a total of 153,216 cores and 217 TB

of memory partitioned on 32GB of NUMA memory per node. The nodes are interconnected by a *Gemini Cray 3D torus* network.

Our tests used real DNA sequences retrieved from NCBI[1]. The sizes of these real sequences range from 1 KBP (Kilo Base Pairs) to 23 MBP (Millions of Base Pairs). The 23000k x 23000k comparison is run only on the HOPPER machine due to the huge sequence size. To understand the speedups presented in this section, it is worth noticing that our SW algorithm uses all the processing elements to calculate a single similarity matrix. Therefore, it involves a considerable amount of inter-processor communication.

4.2.2.4 Results for AMD16 - Small Multicore

On the AMD16 machine, we executed the BSP++ MPI and BSP++ OMP versions. Figures A.9 show the speedups obtained for each version. As expected, the speedups obtained with the BSP++ OMP version are greater than the ones obtained with the BSP++ MPI version, since AMD16 is a shared-memory machine. Also, for the 1K comparisons, the speedups for 16 cores are not good (3.84 for BSP++ MPI and 7.83 for BSP++ OMP). This happens because there is not enough computations to compensate the communication overhead. In this case, it is clearly better to use 8 cores. For the 10K sequences, a speedup of 6.81 was obtained for the BSP++ MPI version and also in this case it is better to use just 8 cores. On the other hand, the BSP++ OMP implementation achieved a speedup of 13.15 for the 10K comparison. Moreover, the comparisons of large sequences (50K, 85K, 150K, 500K and 1000K) achieved speedups that are close to 16, showing that both versions scale up to 16 cores for large sequences.

4.2.2.5 Results for CLUSTER128 - Medium-sized Cluster

In order to see if our implementations are appropriate for a higher number of cores, we executed the same comparisons in the CLUSTER128 machine with BSP++ MPI and BSP++ MPI+OMP. To compute the speedups (Figure A.10), the times for one processor with the BSP++ MPI version were used since, in this case, the *put* primitive is empty and the code basically performs a call to the sequential SW function. For the CLUSTER128 machine, the best speedups were obtained for the BSP++ MPI+OMP version. With this version and 128 cores, a speedup of 116 was achieved for the 1000K comparison, reducing the execution time from 12,487.79s (one core) to 171.56s (128 cores). To do the same comparison with the same number of cores, the BSP++ MPI version achieved a speedup of 73. This difference in speedups happened because the hybrid version takes profit of a two-level communication scheme that matches perfectly the cluster of multi-core architecture.

[1] www.ncbi.nlm.nih.gov

For sequences whose size is smaller or equal to 150K, the speedups were not good for 128 cores. This is an indication that a hybrid MPI/OpenMP solution is an appropriate strategy to compare very large sequences in clusters of multi-cores, being able to achieve very good speedups and, thus, drastically reduce the execution times.

4.2.2.6 Results for HOPPER - Large Cluster

We tested the scalability of our versions on a Petaflops cluster with a very large number of cores for huge sequences sizes. The results shown in figure A.12 present the overall execution time of our versions with sequences of size 5MB and 23MB respectively. Due to the limitation on the reservation time on the `HOPPER` cluster, our tests used from 96 up to $3,072$ cores in the 5MB sequence comparison. For the 23 MB sequence comparison, we used from 384 to $6,144$ cores. The hybrid version used 24 threads OpenMP per MPI process (node). It is worth noticing that the `HOPPER` platform nodes are NUMA and thanks to the SPMD style under the OpenMP paradigm, we were able to get a speedup of 24 on one node (24 cores). The results show that the MPI version presented a linear speedup up to 384 cores and using more cores degrades the performance due to the increase of the communication and synchronization time. On the other hand, the Hybrid version (MPI+OpenMP) presented a close to linear speedup for up to $3,072$ cores, thus decreasing the execution time from $4,595.89$ seconds (96 cores) to 186 seconds ($3,072$ cores).

In order to assess the good scaling of the hybrid version, we used it to compare huge DNA sequences (23 Millions of Base Pairs). Figure A.11 shows the overall execution times for up to $6,144$ cores. The hybrid version was able to achieve an acceleration of 15 from 384 cores to $6,144$ cores, achieving a close to linear speedup. In this case, the execution time was reduced from 24,480 seconds (6 hours and 48 minutes) to 1,599 seconds (26 minutes and 39 seconds). This shows the appropriateness of our hybrid version on a Petaflops architecture.

4.2.2.7 Comparison to state of the art implementations

Table 4.2 lists eleven proposals of parallel SW implementations. We can see that the best speedups are obtained when only the score is provided. On average, the speedups obtained with coarse-grained strategies are better than the ones obtained with fine-grained ones. With the exception of ours, all approaches in this table were designed and optimized for one target platform. Porting them to other platforms would involve a tremendous amount of reprogramming. In the last row, we present the details of our SW parallel approach. We were able to generate code for MPI, MPI&OpenMP and OpenMP. The speedups/GCUPs obtained for all BSP++ SW versions are comparable to the ones reported in the literature, showing that our versions are competitive with existing implementations. Our BSP++ SW versions

were able to compare huge sequences with general-purpose based platforms. On the large scale configuration (`HOPPER` machine), the hybrid version shows an efficiency of 92% in hybrid mode. Maximum GCUPS obtained are 15.5 which are more than most state of the art implementation.

4.3 Conclusion

In this chapter, we implemented a C++ library based on a high-level parallel programming model – BSP – for which we demonstrated an excellent level of performances on various parallel architectures ranging from multi-cores, Cell processors or large scale clusters.

However, the effort required for implementing BSP++ shown that a lot of code is required to support multiple variation of a single code for different parallel architectures. This is apparent as we need a very specific macro to trigger BSP evaluation and to control which and how architecture influence library function calls. Therefore, we think that a better methodology is required to integrate architectural informations into the code generation process of generic code in general and *DSEL*s in particular.

Paper	Platform	Comparison	Grain	Output	MaxSize Compared	# Elements	Best Speedup	GCUPs
State of the Art								
[Boukerche 2009]	cluster (MPI)	seq x seq	fine	score,align.	24,894,250	64 cores	33x	*1.45
[Cehn 2003]	2 clusters (MPI)	seq x seq	fine	score,align.	816,394	20 procs	14x	*0.37
[Rajko 2004]	cluster (MPI)	seq x seq	fine	score,align.	1,100,000	60 procs	39x	*0.25
[Noorian 2009]	cluster (MPI/OpenMP)	query x dbase	coarse	score	2,000	24 cores	14x	*4.38
Our proposal								
	cluster (MPI)	seq x seq	fine	score	1,072,950	128 cores	73x	6.53
	cluster (MPI/OpenMP)				1,072,950	128 cores	116x	10.41
	OpenMP				1,072,950	16 cores	16x	0.40
	CellBE				85,603	8 SPEs	—	0.14
	cluster of CellBE				85,603	24 SPEs	(8:24) 2.8x	0.37
	Hopper(MPI)				5,303,436	3072 cores	260x	3.09
	Hopper(MPI+OpenMP)				24,894,269	6144 cores	5664x	15,5

Table 4.2: Comparative view of the approaches that implement SW in HPC platforms – *Rows in gray highlight the approaches that are similar to ours.*

Chapter 5

Toward Architecture-Aware Libraries

Contents

> *"It's hardware that makes a machine fast. It's software that makes a fast machine slow."*
> — Craig Bruce

Meta-programming techniques allow us to hide from the end-user the code generation process and also abstract the user interface with strong domain semantic but these approaches lack of a methodology to specify complete *DSEL* or Active Libraries. In this chapter, we will discuss such methodology and how we extend it to be suitable for multi-architectural support and how it can integrate elements from high-level abstractions of parallel systems.

5.1 Generative programming

Generative Programming has been defined by Czarnecki in [Czarnecki 1998] as "*a comprehensive software development paradigm to achieving high intentionality, re-usability, and adaptability without the need to compromise the runtime performance and computing resources of the produced software*". This approach consists in defining a model to implement several components of a system. Current practices assemble manually these components. For example, the Standard Template Library provides components that the user needs to aggregate according to his *configuration knowledge*. Generative Programming pushes further this approach by bringing automation in such practices. The model that the developer uses to assemble components is moved to a generative domain model. This results in a generator embedding a *configuration knowledge* that takes care of combining the components. This method can be embedded within a library. These specific libraries are called *active libraries*. Based on the fact that complex software systems can be broken down to

a list of interchangeable components which tasks are clearly identified and a series of generators that combines components by following rules defined by an *a priori* domain specific analysis, Czarnecki proposed a methodology called **Domain Engineering Method for Reusable Algorithmic Libraries** (*DEMRAL*) showing a possible formalization of Generative Programming techniques [Czarnecki 2000]. It relies upon the fact that algorithmic libraries are based on a set of well defined concepts:

- **Algorithms**, that are tied to a mathematical or physical theory;
- **Domain entities and concepts**, which can be represented as abstract data types with container-like properties;
- **Specialized functions** encapsulating algorithm variants depending on data type properties.

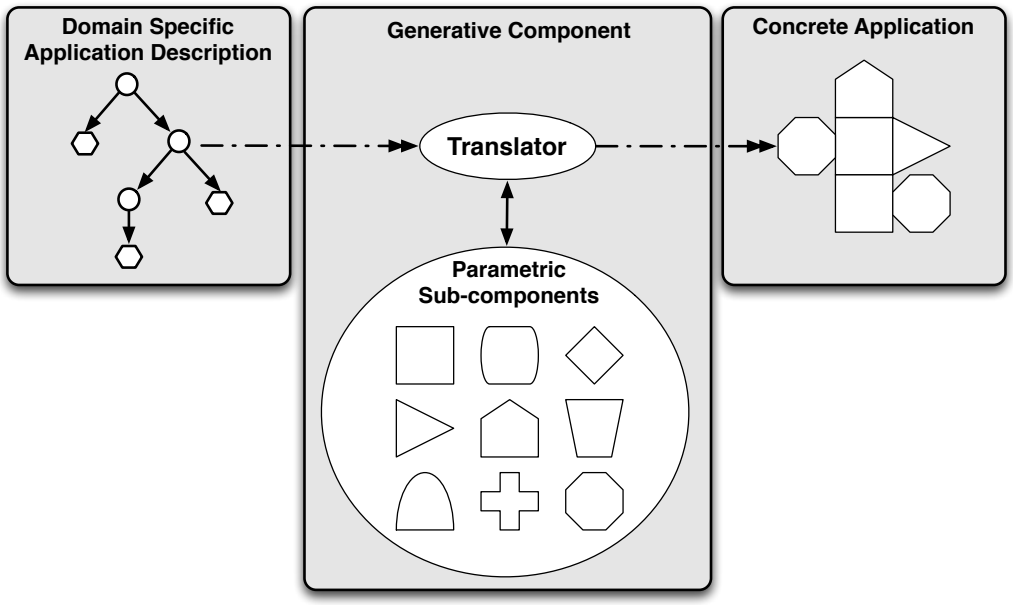

Figure 5.1: The *DEMRAL* methodology

Figure 5.1 illustrates the *DEMRAL* methodology. *DEMRAL* reduces the effort needed to develop software libraries by limiting the amount of code to write. As an example, a library providing N algorithms operating on P different related data-structures may need the design and implementation of $N * P$ functions. Using *DEMRAL* , only N generic algorithms and P data structure descriptions are needed as the code generator will specialize the algorithm with respect to the data structure specificities. This approach allows high re-usability of generic components while their behaviors can be customized.

Taking into consideration the *DEMRAL* methodology helps designing *DSEL*s as this formalization relies on the software aspect of generic components. *DEMRAL* can also be seen as a specialization of paradigms like object-oriented programming, aspect programming or model driven engineering [Gokhale 2003]. The interest for such techniques in our case is that the actual code processing steps can be hidden under a user-friendly interface with domain oriented entities and relationships.

5.2 From *DSEL*s to Architecture Aware *DSEL*

The common factor of all existing *DSEL*s for scientific computing is that the architecture level is mostly seen as a problem that requires specific solutions to be dealt with. The complexity of hand-maintained Expression Templates engines is the main reason why few abstractions are usually added at this level. We propose to integrate the architectural support as another generative component. To do so, we introduce a new methodology which is an hardware-aware extension of the *DEMRAL* methodology.

In this Architecture Aware DEMRAL (*AA-DEMRAL*) methodology, the implementation components are themselves generated from a generative component which translates an abstract architecture description into a set of concrete implementation components to be used by the software generator. In the same way that *DEMRAL* initially removed the complexity of handling a large amount of variations of a given set of algorithms, the Architecture-Aware approach that we propose leverages the work needed for supporting different architectures. By designing a small-scale *DSEL* for describing architectures, the software components used by the top-level code generator are themselves the product of a generative component able to analyze an abstract architecture description to specify the structure of these components. Figure 5.2 illustrates the new *AA-DEMRAL* methodology.

By analogy with the *DEMRAL* methodology, a library providing N algorithms operating on P different related data structures while supporting Q architectures will only need the design and development of $N+P+Q$ software components and the two different generative components (the hardware one and the software one). The library is still designed with high re-usability and its development and maintenance are simplified. Such an approach keeps the advantage of the *DEMRAL* methodology and permits to keep focusing on domain related optimizations while developing the library. In addition, the generic aspect of the components at both levels (hardware and software) allows the generative components to explore a complete configuration space with a sub-part corresponding to specific architectural optimizations thus making the code generation process strongly aware of architectural aspects. The best solution space can then be selected by the generative components.

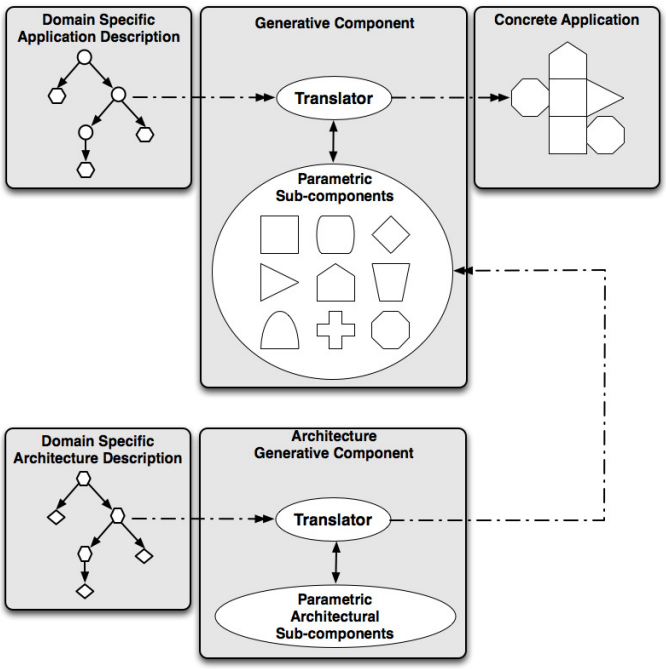

Figure 5.2: The *AA-DEMRAL* methodology

5.3 AA-DEMRAL and Parallel Programming Abstractions

If the principle of our proposed *AA-DEMRAL* methodology seems simple, its practical implementation requires to be able to define a proper set of parametric, architecture dependent, components to be combined by the high-level translator. Our choice is to use **Parallel Skeletons** as a base for those components, the main advantages being:

- **Isolation**: the parallel semantic of skeleton is independent from their actual implementation on a given architecture. This simplify the code generation process as we can enforce that any skeletons required by the application level will have a proper semantic whatever the selected target architecture.
- **Nesting**: as skeletons are defined as higher-order functions, we can exploit skeleton nesting to support hierarchical architectures. Such a support can be implemented by perusing the architecture description to detect and adapt skeleton depending on the current hierarchical level.
- **Adaptivity**: skeletons are – by design – able to support domain specific parallel patterns whenever needed.

The challenges behind such an implementation are:

- Abstract architectural components to specify a hardware configuration;

- Build a generative component that can embed knowledge to choose between hardware components;
- Build a generative component that can embed knowledge to aggregate software and hardware components.

To do so, an actual implementation of an *AA-DEMRAL* based library is then based on a compile-time description of an architecture and a skeleton generation scheme based on function properties.

The elements of architectures our methodology is interested in are two-fold :

- Runtime independent information about the hardware like the presence of a given ISA, the number and size of the register files or the number of super-scalar units. Those informations are accessible through various meta-functions providing these values as integral compile-time constants.
- Software related configuration that describes which – if any – runtime components is required to access a given hardware feature at the software level. As an example, we want to be able to discriminate shared memory hardwares on the fact the software we build need to use OpenMP or Intel TBB. In this case, the architecture is described as a compile-time tag constructed by using the informations given by the compiler through the use of architecture specific options or via runtime specific preprocessor symbols as shown in listing 5.1.

Listing 5.1: Some NT^2 architecture descriptors

```
1  struct cpu_   : unspecified_<cpu_> {}; // cpu_: no special info
2  struct simd_  : cpu_ {};               // simd_: any SIMD architecture
3
4  struct sse2_  : simd_ {};               // architecture supporting SSE2
5  struct avx_   : sse2_ {};               // architecture supporting AVX
6
7  // general shared memory architecture description
8  template<typename BackEnd, typename Core>
9  struct shared_memory_   : Core {};
10
11 // shared memory architecture using OpenMP as runtime
12 template<typename Core>
13 struct openmp_   : shared_memory_<openmp_<Core>, Core> {};
14
15 // shared memory architecture using Intel TBB as runtime
16 template<typename Core>
17 struct tbb_   : shared_memory_<tbb_<Core>, Core> {};
```

For every supported architecture, a descriptor is defined using inheritance to organize related architecture components. In addition to this inheritance scheme, architectures descriptors can be nested (such as `openmp_`). This nesting is computed at compile-time by exploiting information given by the compiler or by user-defined preprocessor symbols. For example, the default architecture computed for a code compiled using AVX and OpenMP is `openmp_< avx_ >`. This

nesting will then be exploited when parallel skeletons will be generated through combination of the OpenMP and AVX versions of each skeleton.

Next to the architecture description, we define a set of function properties that can be used to define **function descriptors**. Those descriptors bind a function name to a type carrying semantic information about how to handle code generation for this function, and thus which skeleton or skeleton combination is required. Those types can be split into two broad families:

- elementwise functions that operate on their arguments at a certain position, without dependencies between operations on different positions. Those functions are usually mapped over a `map` or `mapindex` like skeleton.

- non-elementwise functions, which output can not be combined with an elementwise function but which input is still combinable. Their properties and parallel potential depends on the considered functions. They include reduction and partial reduction functions, scan functions like `cumsum` and external kernels which encapsulate arbitrary parallelism. THe skeletons encountered here can be either `fold` or `scan`

As an example, listing 5.2 presents the descriptors for various functions. `plus` is registered as a classical elementwise operation. `sum` is a reduction and its descriptor defines it as a reduction based on `plus` and `zero`. Then, the matrix-matrix product function is registered as an external kernel.

Listing 5.2: Function descriptors for some NT^2 functions

```
1 struct plus_ : elementwise_<plus_> {};
2 struct sum_ : reduction_<sum_,plus_,zero_> {};
3 struct mtimes_ : unspecified_<mtimes_> {};
```

Each family of functions is then tied to a generic skeleton implementation that will generate the proper skeleton code. Note also that those descriptors could be used to split large expression into sub-expressions with a composable semantic. An example of such split is the combination of elementwise and reduction in a single expression which usually requires the reduction to be evaluated ahead of the elementwise parts.

The last kind of element that our architecture model needs to capture is how different code fragments generated by an arbitrary function combination can be tied together. If the classical way to chain code fragments is to run them sequentially, one architecture may support an efficient way to use a `pipeline` like skeleton to build a task graph of data-oriented skeletons. This task-graph can either be synchronous or asynchronous. In the later case, our architecture descriptor will provide informations on how to spawn and wait on asynchronous tasks.

5.4 Conclusion

Our contribution pushes further the *DEMRAL* methodology and provides a new way for designing architecture aware *DSEL*s . With such an approach, active libraries can take in consideration hardware capabilities and then increase the quality of their code generation process. *DSEL*s keeps their expressiveness and, at the same time, are able to improve their evaluation strategy. This new methodology now requires to be easily implemented in the context of active libraries for parallel architectures. Two such libraries will be presented: Boost.SIMD that takes care of low-level SIMD code generation and NT^2 which is a free-standing implementation of Matlab in C++ . We will explore how *AA-DEMRAL* provides a satisfying model for the skeleton composition and generation. The composability of those libraries will also be demonstrated as NT^2 is able to reuse Boost.SIMD components transparently.

Chapter 6

Boost.SIMD

Contents

> *"He who hasn't hacked assembly language as a youth has no heart. He who does as an adult has no brain."*
>
> — John Moore

In this chapter, we present a first application of our Architecture Aware DEMRAL methodology by building a high-level C++ library providing a generic API for SIMD computations. This use case of our methodology is very interesting as the impact of abstraction overhead on register-level operations can quickly obliterate any potential performance increase, thus making our development strategy paramount to the efficiency of the library code. Work presented in this chapter are the result of Pierre Esterie's PHD thesis [Esterie 2014a] supervision and its associated papers:

- *"N3561 - A proposal to add single instruction multiple data computation to the standard library"* [Estérie 2013]
- *"Boost.SIMD: generic programming for portable SIMDization"* [Estérie 2014b]

which contain more detailed implementation description and additional benchmarks.

6.1 Motivation

Since the late 90's, processor manufacturers provide specialized processing units called multimedia extensions or Single Instruction Multiple Data (SIMD) extensions. The introduction of this feature has allowed processors to exploit the latent data parallelism available in applications by executing a given instruction simultaneously on multiple data stored in a single special register. With a constantly increasing need for performance in applications, today's processor architectures offer rich SIMD instruction sets working with larger and larger SIMD registers (table 6.1). For example, the AVX extension introduced in 2011 enhances the x86 instruction set for the Intel Sandy Bridge and AMD Bulldozer micro-architectures by providing a distinct set of 16 256-bit registers. Similarly, the Intel MIC [Duran 2012] (Many Integrated Core, now known as Xeon Phi) architecture embeds 512-bit SIMD registers. Intel improved AVX with some new instructions and launched AVX 2.0 late 2013. The forthcoming extension from Intel is AVX-512 that will be introduced in the next generation of Xeon Phi, Knights Landing coming in 2014. Using SIMD processing units can also be mandatory for performance on same systems as demonstrated by the NEON and NEON2 ARM extensions [Jang 2011] or the CELL-BE processor by IBM [Kurzak 2009] which SPUs were designed as a SIMD-only system. IBM also introduced in 2012 the QPX [Fox 2011] extension available on the third supercomputer design of the Blue Gene series. QPX works with 32 256-bit registers.

Table 6.1: SIMD extensions in modern processors

Manufacturer	Extension	Registers size & number	Instructions
Intel	SSE	128 bits - 8	70
	SSE2	128 bits - 8/16	214
	SSE3	128 bits - 8/16	227
	SSSE3	128 bits - 8/16	227
	SSE4.1	128 bits - 8/16	274
	SSE4.2	128 bits - 8/16	281
	AVX	256 bits (float only)- 8/16	292
	AVX2 + FMA3	256 bits - 8/16	297
AMD	SSE4a	128 bits - 8/16	231
	XOP	128 bits - 8/16	289
IBM Motorola	VMX	128 - 32	114
	VMX128	128 bits - 128	
	VSX	128 bits - 64	
	QPX	256 bits - 32	
	SPU	128 bits - 128	
ARM	NEON	128 bits - 16	100+
ARM	NEON2		

However, programming applications that take advantage of the SIMD extension available on the current target remains a complex task. Programmers that use low-level intrinsics have to deal with a verbose programming style due to the fact that SIMD instructions sets cover a few common functionalities, requiring to bury the initial algorithms in architecture specific implementation details. Furthermore, these efforts have to be repeated for every different extension that one may want to support, making design and maintenance of such applications very time consuming. Different approaches have been suggested to limit these shortcomings:

- **Intrinsic based solution** The most common way to take advantage of a SIMD extension is to write calls to intrinsics. These low level C functions represent each SIMD instruction supported by the hardware, and while being similar to programming with assembly it is definitely more accessible and optimization-friendly. With a lot of variants to handle all SIMD register types, the set of intrinsics usually only covers functionality for which there is a dedicated instruction, often lacking orthogonality or missing more complex operations like trigonometric or exponential functions. Due to its C interface, using intrinsics forces the programmer to deal with a verbose style of programming. Furthermore, from one extension to another, the Application Programming Interface differs and the code needs to be written again due to hardware specific functionalities and optimizations. Listings 6.1 and 6.2 demonstrate on a small example (a multiply and add operation) the complexity involved by the current programming model and its limitations when an application needs to be portable.

Listing 6.1: Working with SSE4

```
1 __m128i a, b, c, result;
2
3 result = _mm_mullo_epi32(a, _mm_add_epi32(b, c));
```

Listing 6.2: Working with AltiVec

```
1 __vector int a, b, c, result;
2
3 result = vec_cts(vec_madd( vec_ctf(a,0), vec_ctf(b,0), vec_ctf(c,0)),0);
```

- **Compilers** Compilers are now able to generate SIMD code through their auto-vectorizers. This allows the programmer to keep a standard code that will be analyzed and transformed into a vectorized code during the code generation process. Auto-vectorizers have the ability to detect code fragments that can be vectorized. GCC auto-vectorizer [Nuzman 2006] for example is currently available in GCC releases. This automatic process finds its limits when the user code is not presenting a clear vectorizable pattern (i.e. complex data dependencies, non-contiguous memory accesses, aliasing or control flows).

The main approach is to transform the innermost loop-nest to enable its computation with SIMD extensions. The SIMD code generation stays fragile and the resulting instruction flow may be suboptimal compared to an explicit vectorization. Still on the compiler side, code directives can be used to enforce loop vectorisation (`#pragma simd` for ICC and GCC) but the code quality relies on the compiler and this feature is not available in every one of them. Dedicated compilers like ISPC [Pharr 2012], Sierra [Leißa 2014] or Cilk [Robison 2013] choose to add a set of keywords to the language to explicitly mark the code fragments that are candidates to the automatic vectorization process. VaporSIMD [Nuzman 2011] proposes another approach which consists in autovectorizing the C based code to get the intermediate representation of the compiler and then use a Just In Time based framework to generate portable SIMD code. With most of these approaches, the user code becomes non-standard and/or strongly dependent to specific compiler techniques. These techniques also rely on generating SIMD code from scalar code, disregarding the specificities of each of these computing units, including shuffle operations and intra- registers operations.

- **Library based solution Libraries** like Intel MKL [Intel] or its AMD equivalent (ACML) [AMD]. Those libraries offer a set of domain-specific routines (usually linear algebra and/or signal processing) that are optimized for a given architecture. This solution suffers from a lack of flexibility as the proposed routines are optimized for specific use-cases that may not fulfill arbitrary code constraints. In opposition to this "black-box" approach, fine grain libraries like Vc [Kretz 2012] and macstl [PixelGlow Software 2005] propose to apply low level transformations to a specific vector type. For macstl, its support stops at SSE3 and its interface is limited to a few STL-compliant functions and iterators. Vc has a C++ class based approach with support for x86 processors only (SSE to AVX) and provide a list of SIMD enabled mathematical functions.

Due to the factors previously mentioned, providing high level tools able to mix a sufficient abstraction with performance is a nontrivial task that needs to solve important challenges. Several goals are to be faced properly :

- **A generic user interface** The first limitation faced by application developers is the multiplicity of SIMD register types. Furthermore, all the intrinsics are qualified by each data type due to the low level C programming model of such extensions. This restriction forces the programmer to write different versions of the algorithm according to the targets he wants to support. By contrast, a generic approach expresses the algorithms and the data structures as abstract entities. The first challenge of such an approach is to design a high level user interface to keep a strong readability of the code and bury the verbosity of the classic SIMD programming style.

- **C++ standard integration** A lot of existing code relies on the C++ Standard Template Library (STL). Most of them should be able to take advantage of the speedup provided by SIMD extensions. The STL is constructed as a generic library over the following trio of Concepts : Algorithms - Container - Iterators. Switching from a STL code to a fully vectorized version of it must stay straightforward for the user. To accomplish this integration properly, STL Concepts needs to be refined to satisfy SIMD based axioms. On top of that, Boost.SIMD needs to propose a standard like interface with wrappers able to adapt standard components.

- **Effective code generation** The architectural improvements provided by SIMD extensions leads to a significant speedup that we want to reach with Boost.SIMD. Despite the introduction of a generic interface, Boost.SIMD needs to keep the performance of the generated code close to the performance of a "hand written" code. The impact of the generic interface and the code generation engine must be low for the reliability of the library. Specific idioms of SIMD applications need also to be supported.

6.2 Basic Abstractions

Boost.SIMD aims at bridging the lack of proper abstractions over the usage of SIMD registers. This abstraction should not only provide a portable way to use hardware-specific registers but also enable the use of common programming idioms when designing SIMD-aware algorithms. To achieve this, Boost.SIMD implements **an abstraction of SIMD registers** that allow the design of portable algorithms. In addition, a large set of functions are covering the classical set of mathematical functions and utility functions. This section details the components of the library and shows step by step the interface of these components along with their behavior.

6.2.1 SIMD register abstraction

The first level of abstraction introduced by Boost.SIMD is the `pack` class. For a given type `T` and a given static integral value `N` (`N` being a power of 2), a `pack` encapsulates the best type available to store a sequence of `N` elements of type `T`. For arbitrary `T` and `N`, this type is simply `std::array<T,N>` but when `T` and `N` matches the type and width of a SIMD register, the architecture-specific type used to represent this register is used instead. This semantic provides a way to use arbitrarily large SIMD registers on any system and let the library selects the best vectorizable type to handle them. By default, if `N` is not provided, `pack` will automatically select a value that will trigger the selection of the native SIMD register type. Moreover, by carrying informations about its underlying scalar type, `pack` enables proper instruction selection even when used on extensions (like SSE2 and above) that map all integral type to a single SIMD type (`__m128i` for SSE2).

`pack` handles these low-level SIMD register types as regular objects with value semantics, which includes the ability to be constructed or copied from a single scalar value, list of scalar values, iterator or range. In each case, the proper register loading strategy (splat, set, load or gather) will be issued. Listing 6.3 illustrates how the `pack` register abstraction works.

Listing 6.3: Working with `pack`, computing a SIMD register full of 42

```
#include <iostream>
#include <boost/simd/sdk/simd/io.hpp>
#include <boost/simd/sdk/simd/pack.hpp>
#include <boost/simd/include/functions/splat.hpp>
#include <boost/simd/include/functions/plus.hpp>
#include <boost/simd/include/functions/multiplies.hpp>

int main(int argc, const char *argv[])
{
  typedef pack<float> p_t;

  p_t res;
  p_t u(10);
  p_t r = boost::simd::splat<p_t>(11);

  res = (u + r) * 2.f;

  std::cout << res << std::endl;

  return 0;
}
```

`pack` supports multiple constructors. It is copy and default constructible and also supports different methods to initialize a `pack`'s content (loading strategies).

A typedef statement is used before the declaration of the packs for brevity. These declarations include a so-called splatting constructor that takes one scalar value and replicates it in all elements of the pack.

```
p_t u(10);
```

This is equivalent to the constructor on the following line:

```
p_t r = boost::simd::splat<p_t>(11);
```

The user can also initializes every element of the `pack` itself by enumerating them.

```
pack<float> r(11,11,11,11);
```

This constructor makes the strong assumption that the size of the `pack` is correct. Unless required, it is always better to try not to depend on a fixed size for `pack`.

Once initialized, operations on `pack` instances are similar to operations on scalar as all operators and standard library math functions are provided. A simple pattern makes those functions and operators available: if function foo is used, you need to include `boost/simd/include/functions/foo.hpp`. Here, we include `plus.hpp` and `multiplies.hpp` to be able to use `operator+` and `operator*`.

```
res = (u + r) * 2.f;
```

Note that type checking is stricter than one may expect when scalar and SIMD values are mixed. Boost.SIMD only allows mixing types of the same scalar kind, i.e reals with reals or integers with integers. Here, we have to multiply by `2.f` and not simply `2`. We need to keep in mind that fused operations are available for SIMD extensions and in the case of such a statement, we have to generate a call to a fused multiply and add instruction if the targeted extension supports it.

The compilation of the code is rather straightforward: just pass the path to Boost.SIMD and use your compiler options to activate the desired SIMD extension support. For example, on gcc:

```
g++ my_code.cpp -O3 -o my_code -I/path/to/boost/ -msse2
g++ my_code.cpp -O3 -o my_code -I/path/to/boost/ -mavx
g++ my_code.cpp -O3 -o my_code -I/path/to/boost/ -maltivec
```

Some compilers, like Microsoft Visual Studio, don't propagate the fact that a given architecture specific option is triggered. In this case, you need to also defines an architecture specific preprocessor symbol, for example:

```
cl /Oxt /DNDEBUG /arch:SSE2 /I$BOOST_ROOT my_code.cpp
cl /Oxt /DNDEBUG /DBOOST_SIMD_HAS_SSE4_2_SUPPORT /I$BOOST_ROOT
my_code.cpp
```

We can then have a look at the program's output that should look like:

```
{42,42,42,42}
```

Now, let's have a look at the generated assembly code for SSE2:

```
movaps 0x300(%rip),%xmm0
addps 0x2e6(%rip),%xmm0
mulps 0x2ff(%rip),%xmm0
movaps %xmm0,(%rsp)
```

We correctly emitted `*ps` instructions. Note that the abstraction introduced by `pack` does not incur any penalty. Now we can look at the AVX generated assembly:

```
vmovaps 0x407(%rip),%ymm0
vaddps 0x3dc(%rip),%ymm0,%ymm0
vmulps 0x414(%rip),%ymm0,%ymm0
vmovaps %ymm0,(%rsp)
```

We can see that Boost.SIMD generates again the proper AVX code with the call to AVX instructions with `ymm` registers. In the case of Altivec, we want to generate a call to a fused multiply and add operation as it provides such an instruction. The generated assembly code is the following:

```
vspltw v12,v12,0
vspltw v13,v13,0
vspltw v0,v1,0
vmaddfp v1,v12,v13,v0
stvx v1,r10,r9
```

We can see that we correctly splat the data into SIMD registers and then call

6.2.2 Range and Tuple interface

By providing STL-compliant `begin` and `end` member functions, `pack` can be iterated at runtime as a simple container of `N` elements. In addition, the square brackets operator is available on `pack` as `pack` respects the Random Access Container Concept. Similarly, since the size of `pack` is known at compile-time for any given type and architecture, `pack` can also be seen as a tuple and used as a compile-time sequence. Thus, `pack` is fully compatible with Boost.Fusion [de Guzman] and respects the Fusion Random Access Sequence Concept. Listing 6.4 presents the range and Fusion like interface.

Listing 6.4: pack range interface

```
typedef typename pack<float,8> p_t;
float t[] = {0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7};
p_t data(&t[0]); // data = [0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7]

//Random Access Sequence
for(std::size_t i = 0; i<p_t::static_size; i++) data[i] += i;

// Boost Fusion Random Access Sequence
typename boost::fusion::result_of::value_at_c<p_t,0>::type sum;
sum = fusion::accumulate(data, 0.f, add()); // sum = 58.8
```

Another ability of `pack` is to act as an Array of Structures/Structure of Arrays adaptor. For any given type `T` adapted as a compile-time sequence, accessing the i^{th} element of a `pack` will give access to a complete instance of T (acting as an Array of Structures) while iterating over the `pack` content as a compile-time sequence will yield a tuple of `pack` thus making `pack` acts as a Structure of Arrays.

Listing 6.5: pack SOA to AOS

```
using boost::fusion::vector;
using boost::simd::load;
using boost::simd::pack;
using boost::simd::uint8_t;

typedef vector<uint8_t, uint8_t, uint8_t> pixel;
typedef vector<pack<float>, pack<float>, pack<float> > simd_pixel_SOA;
typedef pack< vector<float,float,float> > simd_pixel_AOS;

pixel data[simd_pixel_AOS::static_size]; // [...]

simd_pixel_SOA soa = load<simd_pixel_SOA>(&data[0]);
simd_pixel_AOS aos = load<simd_pixel_AOS>(&data[0]);
```

Line 12, `soa` is loaded with as a Structure Of Array. Each `pack` of the `vector` contains a unique color of pixel as illustrated in figure 6.1.

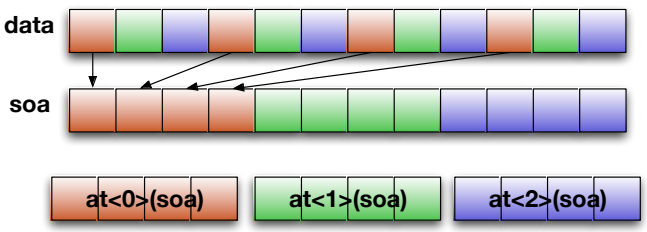

Figure 6.1: Load strategy for SOA

Line 13, `aos` is loaded with as a Array Of Structure. Each `vector` of the `pack` contains a pixel. While accessing `aos` as a compile-time sequence, the i^{th} element of the sequence will yield a `pack` containing a unique color of pixel. Figure 6.2 illustrates this example.

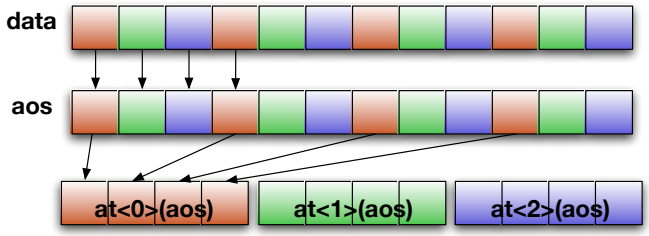

Figure 6.2: Load strategy for AOS

6.2.3 C++ Standard integration

Writing small functions acting over a few `packs` has been covered in the previous section and we saw how the API of Boost.SIMD makes such functions easy to write by abstracting away the architecture-specific code fragments. Realistic applications usually require such functions to be applied over a large set of data. To support such a use case in a simple way, Boost.SIMD provides a set of classes to integrate SIMD computation inside C++ relying on the Standard Template Library (STL) components, thus totally reusing its generic aspect.

Based on Generic Programming as defined by [Stepanov 1995b], the STL is based on the separation between data, stored in various `Containers`, and the way one can iterate these data sets with `Iterators` and algorithms. Instead of providing SIMD aware containers, Boost.SIMD reuses existing STL Concepts to adapt STL-based code to SIMD computations. The goal of this integration is to find standard ways to express classical SIMD programming idioms, thus raising expressiveness and still benefiting from the expertise put into these idioms. More specifically, Boost.SIMD provides SIMD-aware allocators, iterators for regular SIMD computations – including interleaved data or sliding window iterators – and hardware-optimized algorithms.

6.3 SIMD Specific Abstractions

6.3.1 Predicates abstraction

Comparisons between SIMD vectors yield a vector of boolean results. While most SIMD extensions store a $0\backslash\sim0$ bitmask in the same register type as the one used in the comparison, some like Intel Phi or QPX have a special register bank for those types. The Intel MIC has a dedicated 16-bit register to handle the result of the comparison. QPX comparisons put $-1.lf$ or $+1.lf$ inside a QPX register. To handle architecture-specific predicates, an abstraction over boolean values and a set of associated operations must be given to the user. The `logical` class encapsulates the notion of a boolean value and can be combined with `pack`. Thus, for any type `T`, an instance of `pack< logical<T> >` encapsulates the proper SIMD register type

able to store boolean values resulting from the application of a SIMD predicate over a `pack<T>`. Thus, the comparison operators will return a `pack<logical<T> >`. The branching is performed by a dedicated function `if_else` that is able to vectorize the branching process according to the target architecture. Unlike scalar branching, SIMD branching does not perform branching prediction. All branches of an algorithm are evaluated before the result is selected. Listing 6.6 shows a simple example of branching condition with `pack`.

Listing 6.6: Branching example

```
pack<int> a(3), b(1), r;
pack<int> inc(0,1,2,3), dec(3,2,1,0);
r = if_else(inc > dec, a, b); // r = [1,1,3,3]
```

In addition to the classic `if_else` structure, Boost.SIMD provides specific predicate functions that can be optimized. These functions are optimized depending on the types they work with. For example, the `seldec` and `selinc` functions respectively decrement or increment a `pack` according to the result of a comparison and their implementations for integer types rely on a masking technique.

6.3.2 Shuffling operations

A typical SIMD use case is when the user wants to rearrange the data stored in `pack`. This operation is called *shuffling* the register. According to the cardinal of a `pack`, several permutations can be achieved between the data. To handle this, we introduce the `shuffle` function. This function accepts a metafunction class that will take as a parameter the destination index in the result register and return the correct index corresponding to the value from the source register. Listing 6.8 shows such a call.

Listing 6.7: `shuffle` example

```
// A metafunction that reverses the register
struct reverse_
{
  template<class Index, class Cardinal>
  struct apply
       : std::integral_constant<int, Cardinal::value - Index::value - 1> {};
};
[...]
pack<int,4> r{11,22,3,4};
r1 = boost::simd::shuffle<reverse_>(r); // r1 = {4,3,22,11}
```

A second version of the function is also available and allows the user to directly specify the indexes as template parameters:

```
r2 = boost::simd::shuffle<3,2,1,0>(r); // r2 = {4,3,22,11}
```

When called with a metafunction, `shuffle` has the ability of selecting the best permutation strategy available on the target. `shuffle` is implemented to recognize specific patterns that can be mapped to specific intrinsic calls. A generic matcher is able to match a specific permutation that leads to an optimized version of shuffling

operation. The compile-time generic matcher detects such a permutation pattern and `shuffle` dispatches automatically the call to this specific intrinsic. If no specific intrinsics can be called on the targeted architecture, the next choice is to use a general SIMD permutation unit. Such units can perform every permutation. SSSE3 has a special permute unit that permits to arbitrarily permute the values of a register. When SSSE3 is available on the architecture, this unit is used by `shuffle` for performing non optimized permutations through the `_mm_shuffle_epi8` intrinsic. ARM and Altivec also present such permute units. The `shuffle` function uses its generic matcher to detect which call is the best. When the matcher fails to select a specific implementation of `shuffle`, a common version will be called and the permutation will be emulated.

6.4 Benchmark

We present a benchmark of Boost.SIMD applied to image processing by implementing a motion-detection algorithm – Sigma-Delta [Lacassagne 2009]. This algorithm is composed of point-wise operations with two double if-then-else patterns. Its SIMD implementation is not straightforward, as the multiplication and the absolute difference require to promote 8-bit data to temporary 16-bit data or use saturated arithmetic. Its low arithmetic intensity always leads to Memory Bound implementations. Test has been performed using the SSE4.2, AVX, AVX 2.0 and Altivec instruction sets to demonstrate the portability of the library.

The Listing 6.9 shows the Boost.SIMD implementation of the Sigma-Delta algorithm.

Listing 6.9: Boost.SIMD version of Sigma Delta

```
template<class Pixel>
Pixel sigmadelta(Pixel &bkg, const Pixel &fr, Pixel &var)
{
  Pixel diff_img, mul_img, zero=0;
  bkg = selinc( bkg < fr, seldec( bkg > fr, bkg ) );
  diff_img = max(bkg, fr) - min(bkg, fr);

  mul_img = adds(adds(diff_img,diff_img),diff_img);

  var = if_else( diff_img != zero, selinc( var < mul_img
                                         , seldec( var > mul_img
                                                 , var
                                                 )
                                         )
                                 , var
                                 );
  return if_zero_else_one( diff_img < var );
}
```

Table 6.2 details how Boost.SIMD performs against scalar versions of the algorithm. The benchmarks use greyscale images. To handle this format, the type `unsigned char` is used and each vector of the SSE4.2, Altivec or AVX extensions

can carry 16 elements. On the AVX side, the instruction set is not providing a support for this type so Boost.SIMD emulates such a vector but AVX 2.0 supports integer types and can hold 32 elements.

Table 6.2: Results for Sigma-Delta algorithm in *cpp*

Extension	SSE4.2		Altivec	
Size	256^2	512^2	256^2	512^2
Scalar C++(1)	9.237	9.296	14.312	27.074
Scalar C icc	2.619	2.842	-	-
Scalar C gcc	8.073	7.966	-	-
Ref. SIMD(2) JRTIP[Lacassagne 2009]	1.394	1.281	1.380	4.141
Boost.SIMD(3)	1.106	1.125	1.511	5.488
Speedup(1/3)	8.363	8.263	9.469	4.933
Overhead(2/3)	-26%	-13.9%	8.7%	24.5%

The execution time overhead introduced by the use of Boost.SIMD stays below 8.7%. On SSE4.2, it performs better than the SSE4.2 handwritten version while on Altivec, a slow-down appears with images of 512×512 elements. Such a scenario can be explained by the number of images used by the algorithm and their sizes. Three vectors of type `unsigned char` need to be accessed during the computation which is the critical section of the Sigma-Delta algorithm. The 512 KBytes L2 cache of the PowerPC 970FX can not contain the three images in cache. Cache misses becomes preponderant and the Load/Store unit of the Altivec extension keeps waiting for data from the main memory. The L3 cache level of the Nehalem micro-architecture overcomes this problem. The `icc` auto-vectorizer generates SSE4.2 code with the C version of Sigma-Delta while `gcc` fails. The C++ version keeps its fully scalar properties even with the auto-vectorizers enabled due to the lack of static information introduced by the Generic Programming Style of the C++ language.

Figure 6.3 shows the frames per second that Boost.SIMD can obtain against the scalar version of the code on a AVX 2.0 machine. We can see that SSE2 provides an average speedup of $\times 4$ and AVX emulation mode performs significantly better. On the other hand, AVX 2.0 provides good speedups that outperforms other extensions due to its wide registers supporting for 8-bit integers. We can easily see the cache memory effects that impacts the speedups for all extensions while increasing the size of images.

Boost.SIMD keeps the high level abstraction provided by the use of STL code and is able to reach the performance of the vectorized reference code. In addition, the portability of the Boost.SIMD code gives access to the original speedups without rewriting the code. More benchmarks on different applications can be found in

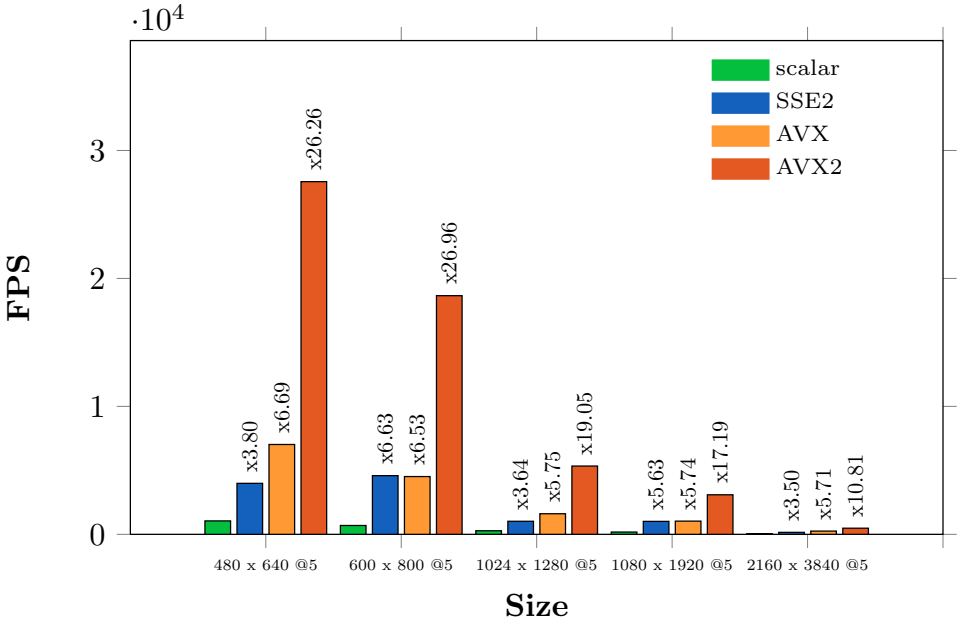

Figure 6.3: Results for Sigma-Delta algorithm on Excalibur

[Estérie 2014b, Esterie 2014a] and demonstrate a similar level of performance.

6.5 Conclusion

Building a library for SIMD extensions with a high level API without loss of performances is not a simple task. Especially when the library needs to be designed in an extensible way for further architecture support. Boost.SIMD demonstrates the applicability of AA-DEMRAL on low-level code generation where overheads can be deadly for performance. We showed that the SIMD register abstraction combined with high level functions makes SIMD computation easy to write and portable over architectures. The API also fits the Standard requirements and is fully compatible with C++ Standard based code. The performance obtained on various benchmarks show that the overhead of the implementation using AA-DEMRAL is minimal, thus validating the low cost of the abstraction.

Chapter 7

The Numerical Template Toolbox

Contents

"Simplicity is prerequisite for reliability."
— Edsger W. Dijkstra

In this chapter, we present another level application of the Architecture Aware DEMRAL methodology by exposing the design and implementation of the Numerical Template Toolbox (NT^2), a C++ library which aims at simplifying the development of high performance numerical computing applications with a multi- architectural support. Work presented in this chapter are the result of Pierre Esterie and Antoine tran Tan PHD thesis supervision and their associated papers:

- *"The numerical template toolbox: A modern C++ design for scientific computing"* [Esterie 2014c]
- *"Automatic Task-based Code Generation for High Performance Domain Specific Embedded Language"* [Tan 2014]

We focus on the API design and application of *AA-DEMRAL* to the implementation of the library. More exhaustive benchmarks and implementation details can be found in said previous publications.

7.1 Motivation

Developing large applications in a simple, fast and efficient way has always been an issue for software developers. For a long time, developers have been limited by cost and availability of computing systems. But in the late 90's, as the yearly increase in CPU frequency started to stall, new ways of using the ever growing number of transistors on chips appeared. SIMD extensions like SSE, Altivec or AVX, multi-processor and multi-core systems, accelerators like GPUs or the Xeon Phi have all reshaped the way computing systems are built. During the same time period, programming methodologies have not changed a lot. Scientific computing applications are still implemented using low-level languages, for instance C or FORTRAN, losing the high-level structure given by the application domain.

Designing **Domain Specific Languages** (or *DSL*) has been presented as a solution to this problem. As *DSL*s allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain, the maintainability and quality of code is increased. One of the most popular examples is MATLAB ™ which provides a large selection of toolboxes that allow a direct expression of high-level algebraic and numerical constructs in a easy to use imperative language.

To accommodate the need of fast prototyping and performances of the end product, some users actually build applications prototypes in a high-level tools and convert them into a low-level language like C or FORTRAN despite the fact that this rewriting may be either costly or cumbersome. Even if those users are scientists, they are not necessarily computer scientists. This fact makes this conversion a tedious, error-prone process. It's also becoming more and more complex as the variety of architectural specificities of the most common computers require those users to master a large amount of different programming models, tools or even languages. For such cases, high-level tools able to simplify development and still able to generate efficient code for any given architecture are more than required if we want parallel computing to become a main stream development tool and process. In this context, we designed a high level library perusing *AA-DEMRAL* methodology to bridge the gap between **High expressiveness** and **Performances**.

7.2 The NT2 Programming Interface

NT^2 has been designed to be as close as possible to the MATLAB language. Ideally, a MATLAB to NT^2 conversion process should be limited to copying the original MATLAB code into a C++ file and performing minor cosmetic changes (defining variables, calling functions in place of certain operators). NT^2 also takes great care to provide numerical precision as close to MATLAB as possible, ensuring that results between versions are sensibly equal. This section will go through the main elements of the NT^2 API and how they interact with the set of supported architectures.

7.2.1 Basic API

The main element of NT^2 is the `table` class. `table` is a template class that can be parametrized by its element type and an optional list of settings. An instance of `table` behaves like a MATLAB multi-dimensional array – including 1-based indexing and column major storage order – and supports the same set of operators and functions. Those operators and functions are, unless specified otherwise, applied to every element of the table, following the standard MATLAB semantic.

NT^2 covers a very large subset of MATLAB functions ranging from standard arithmetic, exponential, hyperbolic and trigonometric functions, bitwise and boolean operations, IEEE related functions, various pattern generators and some statistic and polynomial functions. All those functions support vectorization thanks to BOOST.SIMD[Esterie 2012] (see previous chapter). Moreover, and contrary to most similar library, NT^2 provides support for all real and integral types, both real or complex. Combined with the large set of functions available, this allows NT^2 to be used in a wider variety of domains.

Listing 7.1 shows some NT^2 basic features including the mapping of the colon function (`:`) to the `_` object, various functions, a random number generator and some utility functions like `numel` or `size`.

Listing 7.1: Sample NT^2 code

```
1 table<double> A1 = _(1.0,1000.0);
2 A2 = A1 + randn(size(A1));
3 double rms = sqrt( sum(sqr(A1(_) - A2(_))) / numel(A1) );
```

Listing 7.2 shows the corresponding MATLAB code.

Listing 7.2: Corresponding MATLAB code

```
1 A1 = (1.0:1000.0);
2 A2 = A1 + randn(size(A1));
3 rms = sqrt( sum(sqr(A1(:) - A2(:))) / numel(A1) );
```

The main difference lies in the slight syntax change around the colon function (`:`) that turns into the `_` NT^2 object. Most functions keep their name and interface, allowing for a fast translation between the two languages.

7.2.2 Indexing and data reshaping

Indexing and reshaping of data is one of the main assets of the MATLAB language as it maximizes the expressiveness of array-based expressions. In NT^2, accessing parts of a table is done with `operator()` which handles various indexing values: integer and table of integers, range created by the `colon` function (`_` for short) or contextual keywords like `begin_` and `end_`. Arbitrary extraction, dimension reinterpretation, shifting, and stencil computations can be expressed with that syntax. Listing 7.3 shows how a Jacobi update step can be written using such indexing.

Listing 7.3: Cross stencil for the update step of the Jacobi method with NT^2

```
new_(_(begin_+1, end_-1), _(begin_+1, end_-1))
  = (   old_(_(begin_  , end_-2), _(begin_+1, end_-1))
      + old_(_(begin_+2, end_)  , _(begin_+1, end_-1))
      + old_(_(begin_+1, end_-1), _(begin_  , end_-2))
      + old_(_(begin_+1, end_-1), _(begin_+2, end_  ))
    )/4.f;
```

7.2.3 Linear Algebra support

NT^2 supports the most common matrix decompositions, system solvers and related linear algebra operations via a transparent binding to BLAS and LAPACK. MATLAB syntax is preserved for most of these functions, including the multi-return for decompositions and solvers or the various options for customizing algorithms. The QR decomposition of a given matrix `A` while retrieving the decomposition permutation vector is done this way:

```
tie(Q,R,P) = qr(A,vector_);
```

which can be compared to the equivalent MATLAB code:

```
[Q,R,P] = qr(A,'vector');
```

The `tie` function is optimized to take care of maximizing the memory reuse of output parameters so the minimal amount of copies and allocations are performed.

7.2.4 Compile-time Expression Optimization

Whenever a NT^2 statement is constructed, potential automatic rewriting may occur at compile-time on expressions for which a high-level algorithmic or an architecture-driven optimization is possible. This compile-time expression optimization is similar to the one introduced in BOOST.SIMD. Considered optimizations include:

- Fixed-point transformations like `trans(trans(x))` or other functions combinations that can be precomputed as being equivalent to a simpler function;
- Fusion of operations like `mtimes(a, trans(b))` which can directly notify the GEMM BLAS primitive that `b` is tranposed;
- Architecture-driven optimizations like transforming `a*b+c` into `fma(a,b,c)`.
- Sub-matrix access like `a(_,i)` into an optimized representation enabling vectorization.
- Inter-statement loop fusion using the `tie` function to group statements of compatible dimensions in a single loop nest.

7.3 Implementation

7.3.1 Parallel code generation

NT^2 is an Expression Template based *DSEL* that uses Boost.Proto [Niebler 2007] and Boost.SIMD (see chapter 6). Boost.Proto is used as its expression template engine and replaces the classical direct walk-through of the compile- time AST done in most C++ *DSEL*s by the execution of a mixed compile-time/runtime algorithm over a Boost.Proto standardized AST structure. The expression evaluation strategy of NT^2 is driven by the *AA-DEMRAL* methodology introduced in chapter 5. This code generation process is based on three steps:

- As compile-time ASTs are built, optimizations of the AST structure are performed. In this process, patterns of functions and operators calls that can be replaced by a more efficient implementation are caught and regenerated using architecture-driven rules sets. Two nodes are said to be nestable if their code can be generated in a single loop nest. If two nodes are not nestable, the most complex one is replaced by a temporary terminal reference pointing to the future result of the node evaluation. The actual sub-tree is then scheduled to be evaluated in advance, providing data to fill up the proxy reference in the original tree. As an example, figure 7.1 shows how the expression `A = B / sum(C+D)` is built and split into sub-ASTs handled by a single type of skeleton.

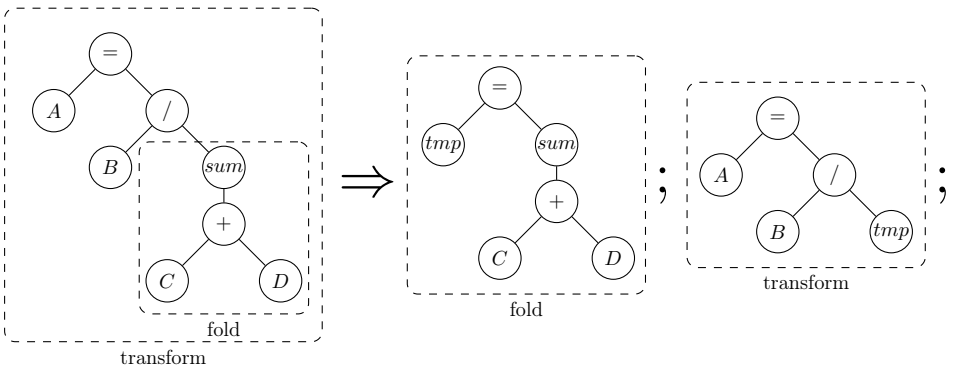

Figure 7.1: Parallel Skeletons extraction process

- The NT^2 code generator then generates successions of loop nests based on the top level AST node descriptor. This turns the original AST into a list of dependent sub-ASTs, each being tied to a single **Parallel Skeletons** [Cole 2004]. Even if a large number of skeletons have been proposed in the litterature [Kuchen 2002, Ciechanowicz 2010], NT^2 mainly uses three data-oriented skeletons: **transform**, **fold** and **scan**.

- Each ASTs are then evaluated into a proper loop nest code. The NT^2 expression evaluation is based on the possibility to compute the size and value type. This size is used to construct a loop, which can be parallelized using arbitrary techniques, which then evaluates the operation for any position p, either in scalar or in SIMD mode. The main entry point of this system is the `run` function that is defined for every function or family of functions. `run` takes care of selecting the best way to evaluate a given function in the context of its local AST and the current output element position to compute. At this point, NT^2 exploits the information about the function properties and dispatch to a specific loop nest generator for each family of functions (elementwise, reduction, etc). NT^2 then uses the nestability of parallel skeletons to call the SIMD and/or scalar version of each skeleton involved in a series of statements to recursively and hierarchically exploit the target hardware. At the end of the compilation, each NT^2 expression has been turned into the proper series of nested loop nests using combinations of OpenMP, SIMD and scalar code. Each of these skeleton is a NT^2 function object.

7.3.2 Support for Asynchronous Skeletons

As sketched above, once a DSEL statement has been issued either by the user or as a list of temporary statements generated by an AST split, they are executed following the simple fork-join model enforced by OpenMP and TBB. As the number of statements grows, the cost of synchronization, temporary allocation and cache misses due to poor locality handling lower the performance.

Our approach is to use the automatic AST splitting system to derive a dependency graph between those statements and turn this graph into a runtime managed list of coarse grain tasks. In order to exploit inter-statement parallelism, NT^2 requires a runtime that allows a proper level of performance, supports nestability and limits the cost of synchronization. *Tasking* [Ayguadé 2009] or *asynchronous programming* is a such a model. Available in several projects relating to task runtimes such as TBB [Reinders 2010], OmpSs [Ayguadé 2009], HPX [Kaiser 2009], Quark [Yarkhan 2011] or OpenMP (3.0 and 4.0 specifications) [OpenMP Architecture Review Board 2013], this model is able to generate and process an arbitrary task graph on various architectures while minimizing synchronization. The second point is the nestability. To keep the NT^2 skeleton high level model, we need to use an implementation of tasking supporting such composable calls. Traditionally, low-level thread-based parallelism often suffers from a lack of composability as it relies on procedural calls that only work with a global view of the program. Another interface for such a tasking model is the *Future* programming model [Friedman 1976, Baker Jr 1977] that has been integrated by the 2011 C++ Standard [The C++ Standards Committee 2011]. A *Future* is an object holding a value which may become available at some point in the future. This value is usually the result of some other computation but is usually

created without waiting for the completion of the computation. Futures allow for composable parallel programs as they can be passed around parallel function calls as simple value semantic objects and have their actual contents be requested in a non-blocking or blocking way depending on context.

As the AST list generated during the splitting step explicitly describes the data dependencies of the original expression, we can use any kind of asynchronous threading API to execute all tasks in proper sequence. Each of the tasks is launched as a separate thread, generating an instance of a Future which represents the expected result of each of the tasks. Once set up, this graph of Future can be composed either sequentially or in parallel:

- **Sequential composition** is achieved by calling a Future's member function f`.then(`g`)` which attaches a given function g to the Future object f. Here, this member function returns a new Future object representing the result of the attached continuation function g. The function will be (asynchronously) invoked whenever the Future f becomes ready. Sequential composition is the main mechanism for sequentially executing several tasks, where this sequence of tasks can still run in parallel with any other task.
- **Parallel composition** is implemented using the utility function `when_all(` $f1$`, ` $f2$`, ...)` which returns yet another Future object. The returned Future object becomes ready whenever all argument Future objects $f1$, $f2$, etc. have become ready. Parallel composition is the main building block for fork-join style task execution, where several tasks are executed in parallel but all of them must finish running before other tasks could be scheduled.

We use these composition facilities to create task dependencies which mirror the data dependencies described by the generated AST. Here, the Future objects represent the terminal nodes and their combination represents the edges and the intermediate nodes of the AST.

7.3.3 Integration in NT^2

The NT^2 integration is done by:

- **Providing a generic implementation of Futures**. Although NT^2 uses HPX as a prime backend for task parallelism, most systems tend to use runtimes like OpenMP or TBB. Thus we implement a Future class template that acts as a generic template wrapper which maps the current runtime choice to its proper task implementation and related functions.
- **Implementing skeletons for taskification**. NT^2 skeletons have been implemented so their internal implementation relies on Futures and asynchronous calls. To do so, NT^2 skeletons now use a task-oriented implementation by using a *worker/spawner* model. The `worker` is a function object containing a

function call operator that takes a range as parameter and processes it with all the possible optimizations. The `spawner` is a function template which acts as the parallel skeleton: it invokes multiple workers by binding them appropriately to tasks depending on the kind of skeleton required in a given statement.

- **Adding task management to NT^2**. Last part of this implementation lies in the process of chaining the asynchronous tasks generated by the various skeletons spawned from a given set of ASTs. This is done by implementing a Future-based **pipeline** skeleton that explicits the different dependencies required for the evaluation of expressions. Pipelines are then created between temporary ASTs and between sub-slices of pre-existing arrays.

Figure 7.2 shows the final task layout of the simple `A = B / sum(C+D)` expression. The expression is parallelized using both the *worker/spawner* model to take advantage of the data-oriented parallelism inside the array evaluation, and *pipelines* between the two sub-AST generated by the split skeleton.

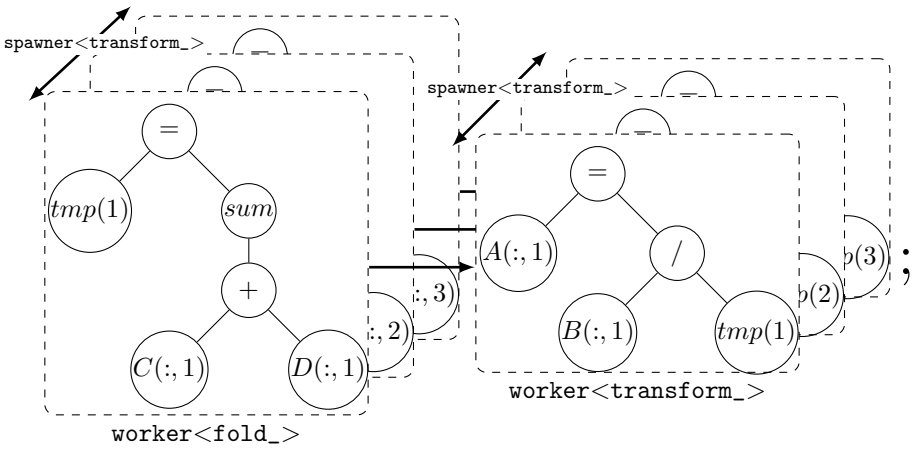

Figure 7.2: Taskification of an AST – *Previous compile decomposition into multiple statements is augmented with the insertion of an asynchronous pipeline between the auto-generated statements.*

The benefits of this system are:

- Instruction Level Parallelism is maintained by using SIMD-optimized workers when it is possible, thus delivering proper performance from the data-parallel layer. This can be seen as a kind of deep nesting where code inside asynchronous tasks are themselves implemented using SIMD based skeletons.

- Optimization across statement boundaries are made possible. They increase data locality, thus ensuring optimal memory accesses. Such optimizations are often difficult to perform with classical Expression Templates as they can only statically access the statement's structure.

7.4 Benchmarks

As for BOOST.SIMD, we will focus on a single application as a benchmarks. More benchmarks with similar behaviors can be found in [Estérie 2014b]. We again implement the Sigma-Delta algorithm as its specificities in term of performance are a challenge for NT^2, due to its multi-statement implementation. A simple implementation of the Sigma-Delta algorithm is given in listing 7.4.

Listing 7.4: Sigma-Delta NT^2 implementation

```
background = selinc( background < frame
                   , seldec( background > frame, background )
                   );

diff     = max(background, frame) - min(background, frame);
sigma3   = muls(diff,uint8_t(3));

variance = if_else( diff != uint8_t(0)
                  , selinc( variance < sigma3
                          , seldec(variance > sigma3, variance)
                          )
                  , variance
                  );

detected = if_zero_else_one( diff < variance );
```

The code structure and the actual functions called are equivalent with the one used in the BOOST.SIMD implementation. The main difference is that the basic block of processing is not the pixel or SIMD pack of pixels but full image frames passed as NT^2 `tables`. Note that, as in MATLAB, constants have to be properly typed and how every step of the algorithm is written without explicit loops.

The NT^2 implementation of Sigma Delta with 8-bit unsigned integers using saturated arithmetic is given in figure 7.3.

As the algorithm is designed to work with unsigned 8-bit integers, the code cannot take advantage of AVX and thus has only been tested on **Mini- Titan** (see figure 7.3). With many load and store operations, the strong scalability of the algorithm can not be preserved. When SSE is enabled, both versions (single-threaded-and multi-threaded) of the code increase their efficiency until hitting the maximum of the memory bandwidth. The SIMD only version is one cycle slower than the handwritten optimized one. This loss comes from very fine grain optimizations introduced in the code. Typically, the difference image does not need to be stored when working with an outer loop on the current frame being processed (C version). The Sigma-Delta implementation shows that the code generation engine of NT^2 leads to a proper optimized version of the application.

7.5 Conclusion

Designing a high level programming tool for High Performance Computing is not an easy challenge. We show that NT^2 has a straightforward and expressive

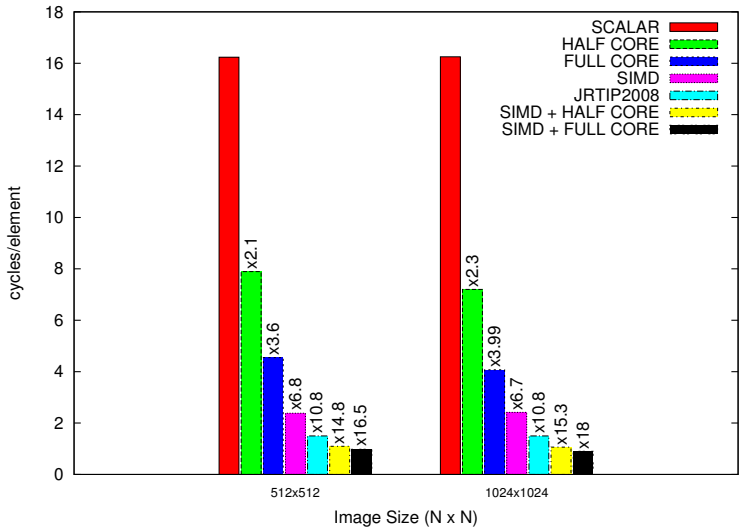

Figure 7.3: Sigma Delta Results

API that guarantees a high level of abstraction. While keeping expressiveness at its maximum, NT^2 takes advantage of the architecture informations to deliver a high level of performance. It allows portability over various architectures and provides a systematic way of implementing new architectural supports. Its generic internal framework permits an easy extensibility of the *DSEL* . Our benchmarks show that NT^2 is able to deliver performance within the range of state of the art implementation.

NT^2 uses expression template techniques and generative programming. It also relies on Boost.Dispatch and Boost.SIMD. Its main contributions are the following:

- Generative Programming helps implementing more flexible scientific computing software with a very high level of abstractions and high efficiency.
- Generic programming inside NT^2 permit an easy multi-architectural support for today's architectures.
- The benchmarks show an efficient code generation system.

Designed as an active library, NT^2 proposes a solution for the design of high level programming tools with multi-architectural support.

Chapter 8

Conclusion and Perspectives

"That's the thing about people who think they hate computers. What they really hate is lousy programmers."
— Larry Niven

Modern scientific challenges have outgrown the capability of scientists to experiment and test their theoretical models, thus forcing them to rely on numerical simulations or data analytics. To satisfy the need of computing power of those scientific codes, the design of computing systems ended up providing extremely efficient machines but with a complexity that made scientist unable to take full advantage of the computing power they were promised. To solve these issues, works have been pursued on how to simplify the design, implementation and maintenance of complex, highly parallel code. Those solutions, however, were either limited to a subset of proprietary hardware, abstract but not efficient or efficient but reserved to experts in parallel computing.

A promising design choices have been the exploration of **Domain Specific** solutions. Those solutions are able to extract valuable information and expertise from an application domain and cannibalize it inside easy-to-use programming tools. This is the way we chose to explore and extend.

First, we examined how and which high-level abstraction for parallel programs could prove useful. Parallel Skeletons and Future-based programming demonstrated the most flexibility and portability across platforms by capturing fundamental aspects of parallel softwares like structured patterns of computations or asynchrony while preserving composability. We demonstrated that modern design idioms for compiled languages like C++ were able to deliver high-level of performances for tools based on those models. Expression Templates and other *DSEL* related techniques allowed us to provide an arbitrary abstract API while generating code close to hand-written code.

While experimenting those techniques, we found out that the effort required to port those tools to a large selection of hardwares could be arbitrary large if no method is used to rationalize those developments. Our work on an Architecture-Aware design for *DSEL*s made obvious the fact that, like regular languages, *DSEL*s need hardware descriptors to be properly optimized and portable.

We showed that, once this methodology was formalized, porting our tools on a large selection of parallel systems was possible within a reasonable time-frame. Along the on-going adaptation of our tools to new architectures, new research directions are still untouched.

Exploration of Irregular Parallelism Scientific computing also contains applications which relies on complex, irregular data structure that require a non-trivial, non-contiguous processing [Aneja 2009]. Algorithms dealing with graphs, trees and other higher-level data structures [Lacassagne 2011] present a clear challenge for Parallel Skeleton. Integrating such irregular patterns into our system would help the design and portability of such applications within the *AA-DEMRAL* framework.

Extension to Data Analytics Numerical computing has been our focus because of its large set of applications. Currently, Data Sciences are becoming as preeminent than HPC once was. Applying our technology to domain like machine learning, computer vision or knowledge discovery techniques could yield interesting results for both communities: defining new scalable parallel algorithms for irregular or data-driven problems on one side; providing highly-efficient and portable data-centric softwares on the other side.

Support for Dynamic Languages Dynamic and scripting languages are becoming increasingly popular. Their applicability to a large selection of applications is backed up by an low entry level for non-experts. The challenge is to find a way to integrate techniques for efficient code generation from within the language. Work is currently being done to integrate SIMD computation support in JavaScript [McCutchan 2014] and Python [Guelton 2014] but are still limited or relying on an external compiling process. A mixed approach – part library, part interpreter solution – is probably interesting to explore.

Support for Reconfigurable Hardware Our work focused on building the software with best performance for a given set of architectures. The same techniques could be used to define the best hardware from a given domain specific code. By using introspection on arbitrary AST of scientific computing code and by targeting languages like System C or OpenCL, we could synthesize hardware on a FPGA like architecture. Using one of the OpenCL FPGA back-end, the cost of generating the required environment to run such a hardware is lessened and fit our multi-stage programming model. The challenge will be to see how far domain specific optimizations can be lead and what kind of impact, as hinted in [Ye 2013], they could have on important hardware metric like power consumption or chip size.

Finally, the main lesson of this work is that injecting high level, domain specific information into the global compilation process can generate better optimizations that the classical collection of low-level, post-intermediate representation based optimizations. However, the amount of work required by meta-programming and *DSEL*s design in C++ limits this approach to a small clique of specialists. Moreover, some optimization opportunities are missed because compile-time *DSEL*s can not access informations like variable name or basic block partitioning. This kind of informations are trivially usable from within a compiler. The idea should be to explore how to make compilers aware of our type-based optimizations and have them provide language extensions for AST handling from within C++ meta-program. Other approaches like compiler scripting are also possible and could be mixed with template based code generation. At a more general scope, exploring how to adapt existing language like C++ to support *DSEL*s design as a first class citizen is our goal for a not so distant future. C++ is currently evolving in new and innovative way and integrating *DSEL*s as part of the language – either as a component on its own or as a collection of introspection related features – is a deep subject which, we think, may bring interesting result to both community.

Appendix A

BSP++ Benchmarks results

A.1 APMC Benchmarks

This section presents the comprehensive list of benchmark figures for the BSP++ APMC implementation presented in section 4.2.1.

For reference, the hardware configurations used were:

- The first machine, the AMD machine, is a quad-core quad- processor 2GHz AMD Opteron 8354 machine with 4 × 2-MB L3 cache and 16-GB of RAM, running the 2.6.26 Linux kernel and g++ 4.3 with O PEN MP 2.0 support and openMPI. In all our experiments, the task/core mapping was set using the sched affinity system call to prevent thread migration between cores and get stable perfor- mance.
- The second machine, the CLUSTER machine, is a cluster of the GRID5000 platform [34]. We used between 2 and 64 nodes connected by a 2 × BCM5704 Gigabit Ethernet Network. Each node is a dual-core bi-processor 2.6-GHz AMD Opteron 2218 with 2 × 2-MB L2 cache and 4-GB of RAM, using the MPICH2.1.0.6 library.

For the sake of understanding, the results are presented using the slowdown metric. The slowdown for a specific computation on a given machine is defined as the execution time of the n-cores machine for the computation multiplied by the number of cores n. Using this metric, the overall execution time for a n-core machine remains constant and achieves a linear speedup. When the parallel efficiency decreases while the number of cores increases, the overall execution time increases. With a super linear speedup, the overall execution time decreases when the number of cores increases.

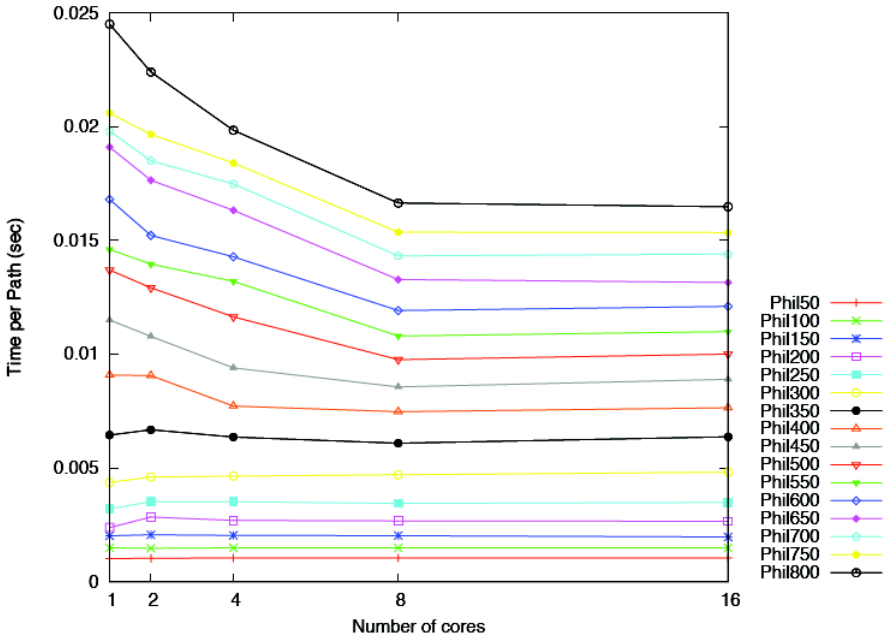

Figure A.1: MPI BSP++ results for the Dining Philosophers (AMD)

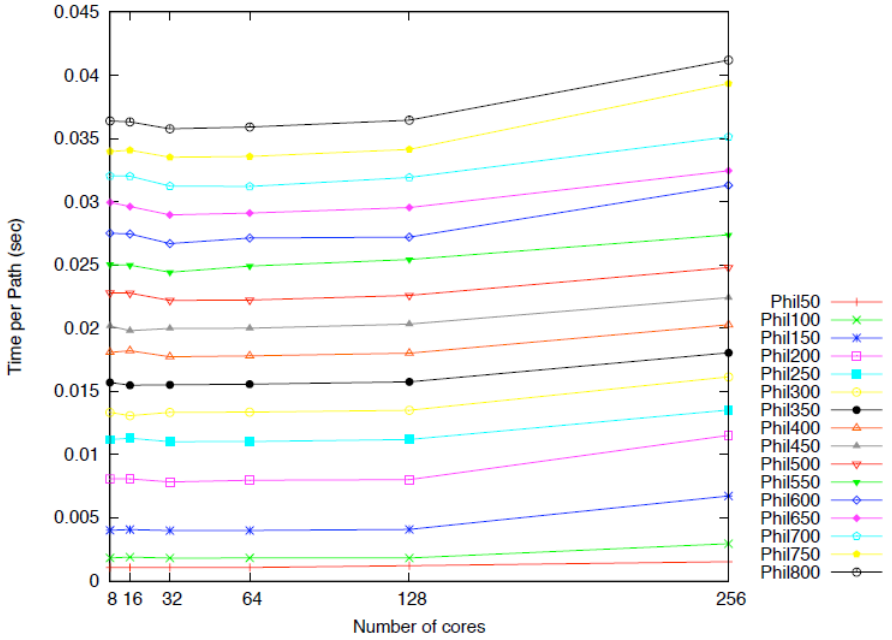

Figure A.2: MPI BSP++ results for the Dining Philosophers (CLUSTER)

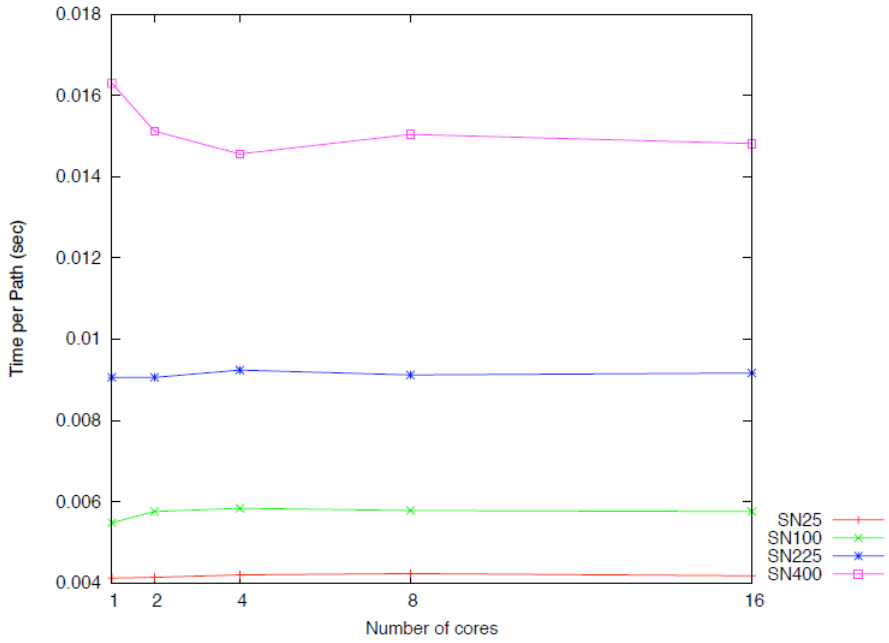

Figure A.3: MPI BSP++ results for the Sensor Network (AMD)

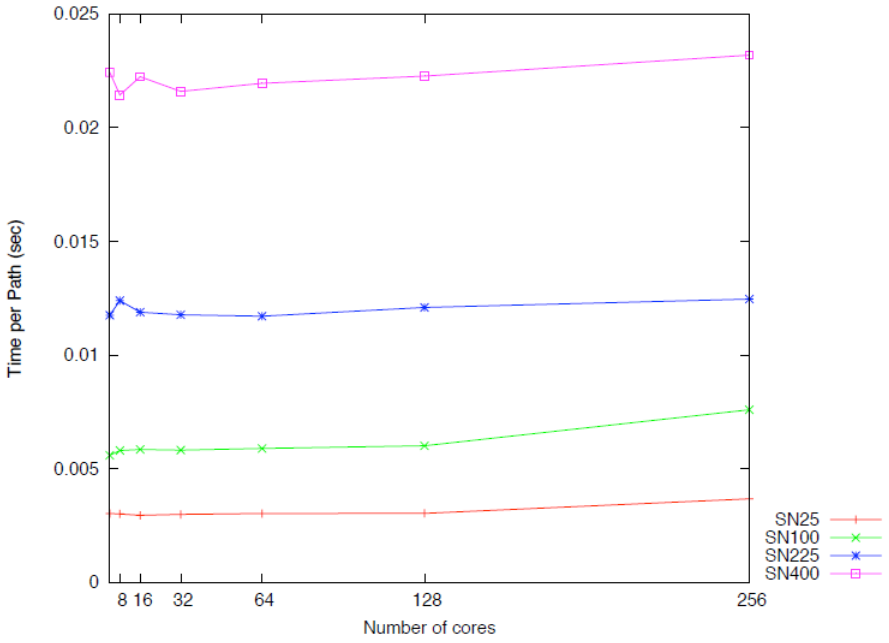

Figure A.4: MPI BSP++ results for the Sensor Network (CLUSTER)

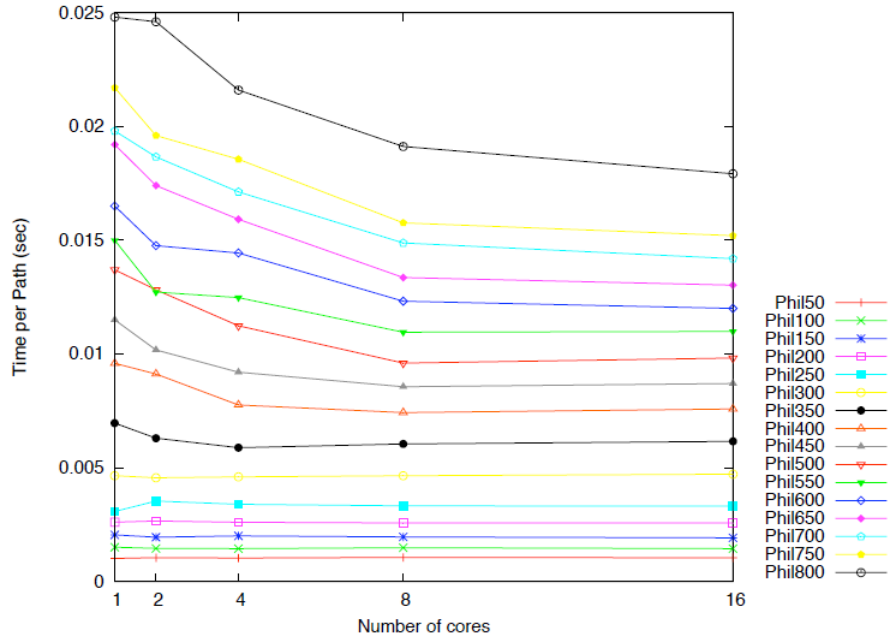

Figure A.5: OpenMP BSP++ results for the Dining Philosophers (AMD)

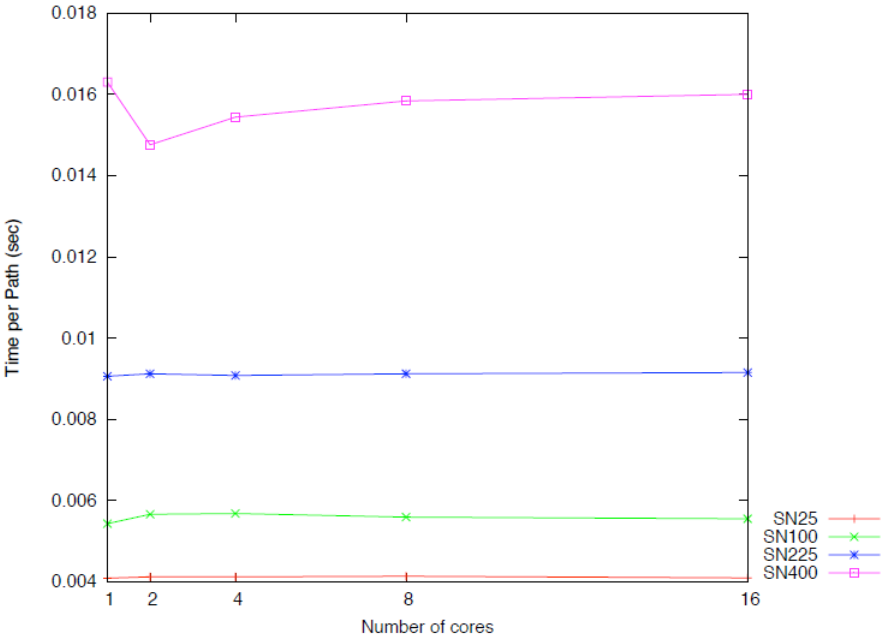

Figure A.6: OpenMP BSP++ results for the Sensor Network (AMD)

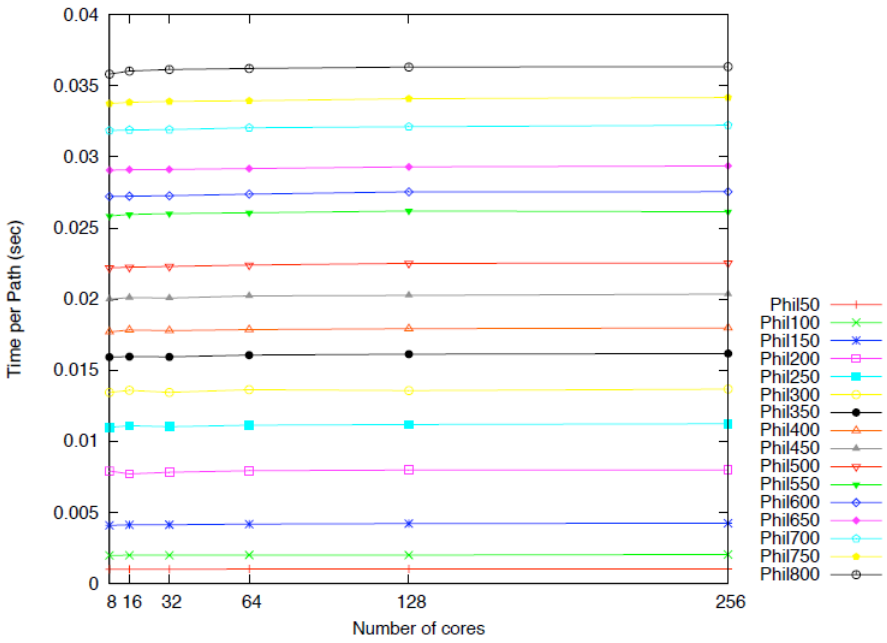

Figure A.7: Hybrid BSP++ results for the Dining Philosophers (AMD)

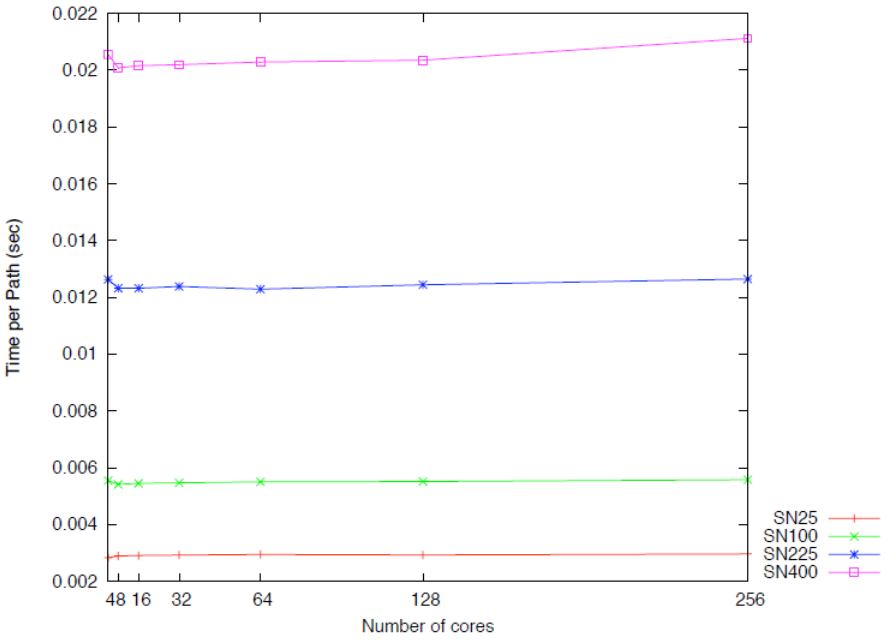

Figure A.8: Hybrid BSP++ results for the Sensor Network (AMD)

A.2 Smith-Waterman Benchmarks

his section presents the comprehensive list of benchmarks figures for the BSP++ APMC implementation presented in section 4.2.2.

For reference, the hardware configurations used were:

- `AMD16`: quad-processor quad-core 2GHz AMD Opteron, 16-GB of RAM and a 3-MB L3 cache running the 2.6.28 Linux kernel and g++ 4.4 with OpenMP 2.0 support and OpenMPI 1.4.2. In all our experiments, the task/core mapping was set using the `sched affinity` system call to prevent thread migration between cores and get stable performance.
- `CLUSTER128`: a 32-node cluster from the Bordeaux site of the GRID5000 platform [Cappello 2010]. Each node is a bi-processor bi-core 2.6-GHz AMD Opteron, 4-GB of RAM and a 2-MB L2 cache using g++-4.4 with OpenMP 2.0 support and the OpenMPI 1.4.3 library.
- `HOPPER`: a 6384-node Cray XE6 cluster, where each node is composed of 24 cores (2 x 12 AMD 2.1-GHz), with a total of 153,216 cores and 217 TB of memory partitioned on 32GB of `NUMA` memory per node. The nodes are interconnected by a *Gemini Cray 3D torus* network.

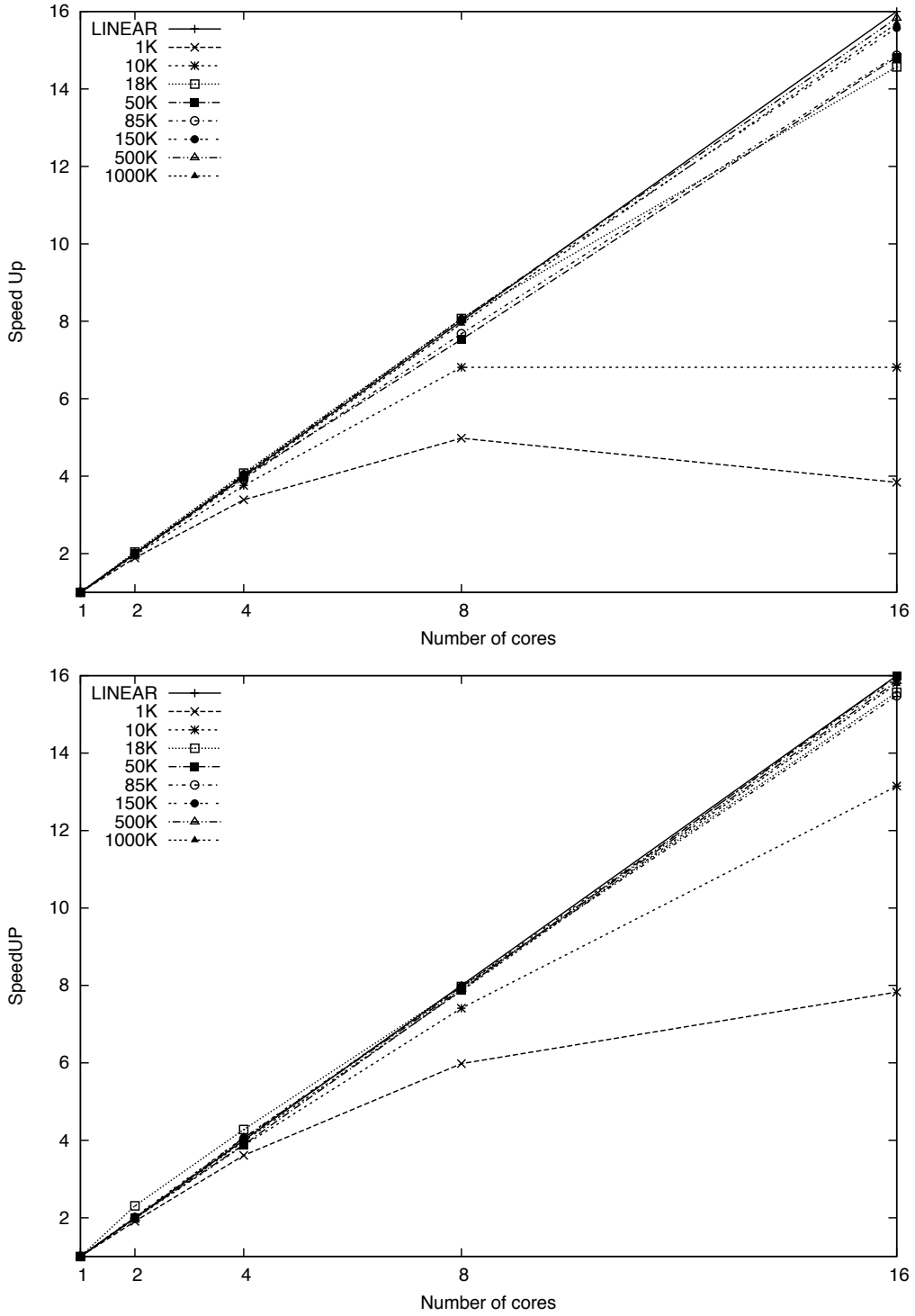

Figure A.9: Speedups for BSP++ MPI and BSP++ OpenMP on AMD16.

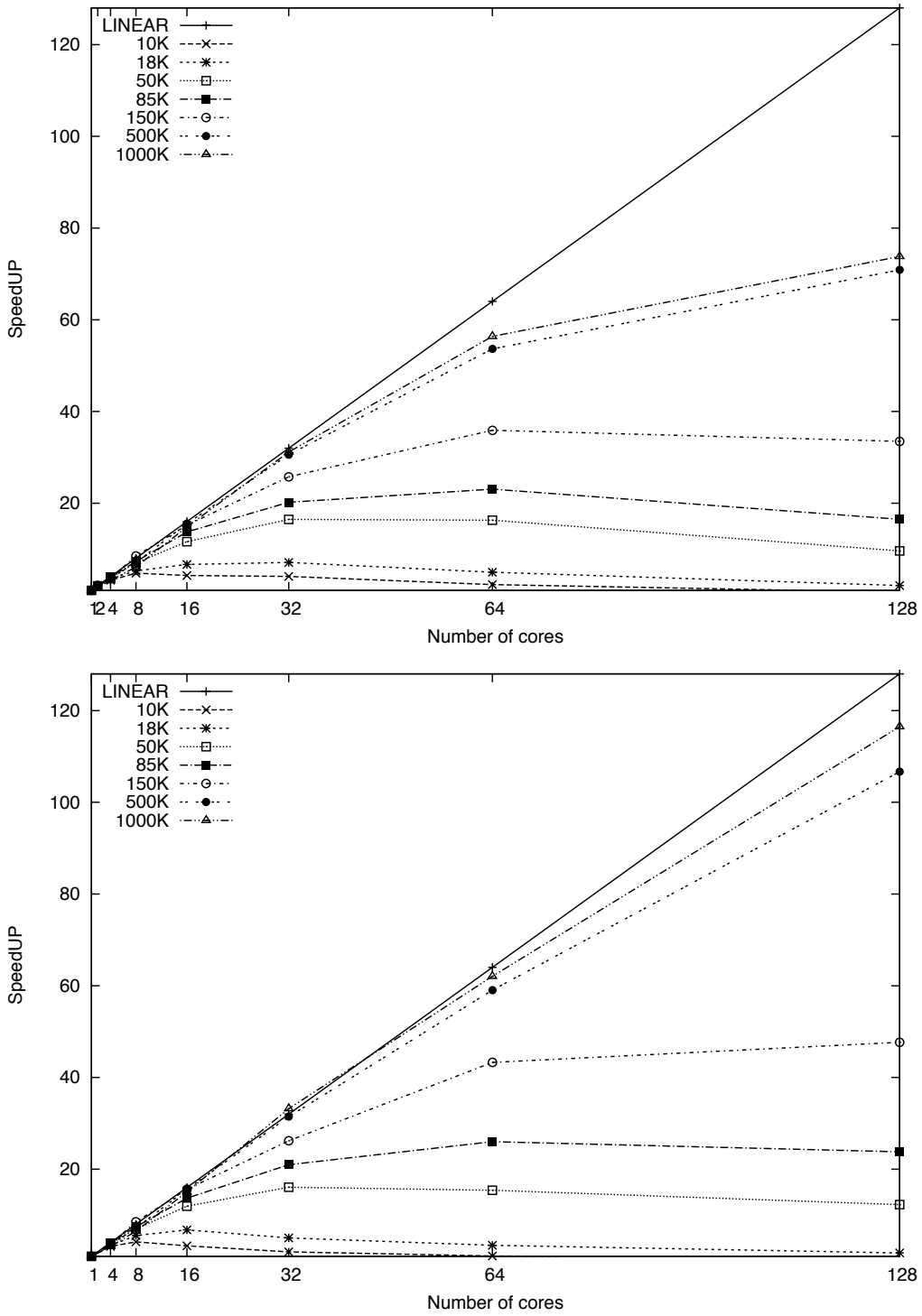

Figure A.10: Speedups for BSP++ MPI and BSP++ MPI+OMP versions on CLUSTER128.

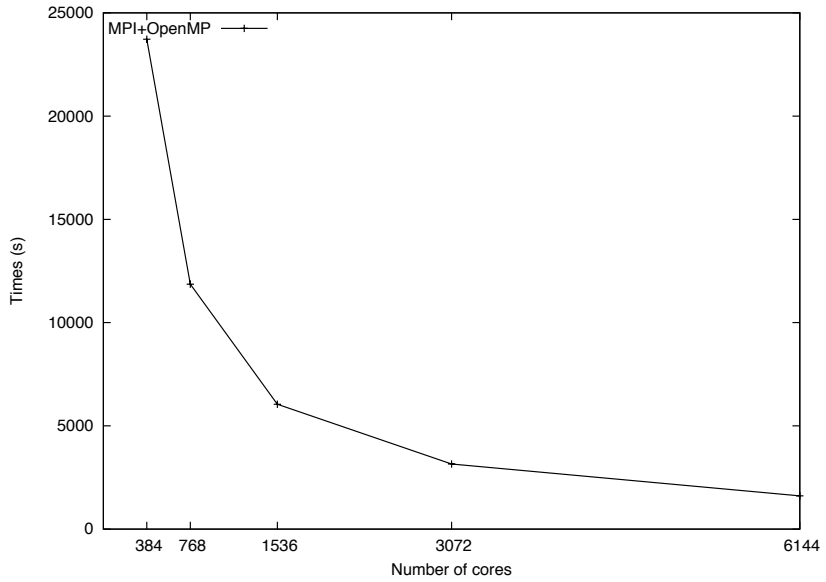

Figure A.11: Overall execution time with 23MB sequences on HOPPER

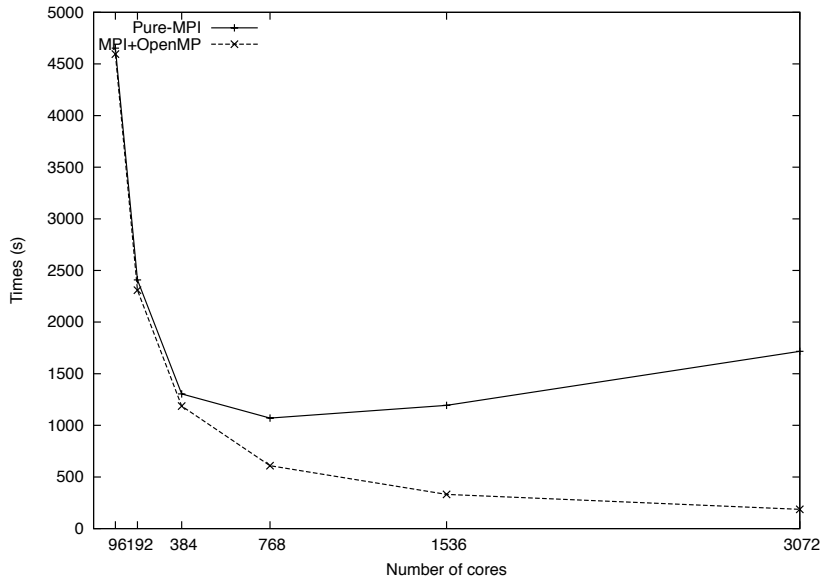

Figure A.12: Overall execution time with 5MB sequence on HOPPER.

Appendix B

Curriculum Vitae

This appendix contains the current version of my Curriculum Vitae.

Joel Falcou

Personal Informations

Born:	2 mai 1980
In:	Sarlat, France
Citizenship	French
MArital Status	Maried, two children
Web page	`www.lri.fr/~falcou`

Contact

Address:	8 Grande Rue
	91940
	Saint Jean de Beauregard
	France
Phone	01 69 20 65 46
Email	`joel.falcou@lri.fr`

Work Experience

2008-Now Assistant Professor, Université Paris-Sud 11
Researcher in the Parallel Architecture team at LRI, computer science engineering teacher at PolyTech Paris Sud,
▷ *Address: Faculté des sciences, F-91405 Orsay Cedex,France.*
▷ *Supervisor : Pr. Daniel Etiemble.*

2012-Now Scientific Advisor, NumScale SAS
Scientific counseling, R&D lead.
▷ *Address: 86 rue de Paris, 91405 Orsay, France.*

April-July 2011 Visiting Professor, PARASOL Team - Texas A&M University
Collaboration around the design and implementation of the STAPL library
▷ *Address: College Station, Texas, TX 10012, USA.*
▷ *Host : Pr. Lawrence Rauchwerger.*

2007-2008 Research Engineer, Université Paris Sud 11
Post-doctoral fellowship. Took part in the OCELLE and TERAOPS French National Research Agency projects
▷ *Address: IEF, Rue Ampère, bâtiment 220-221, 91405 Orsay, France.*
▷ *Supervisor : Dr. Lionel Lacassagne.*

2006-2007 Temporary Teaching and Research Assistant, Université Blaise Pascal - Clermont 2
Researcher in the Architecture team at LASMEA. Teaching computer science engineering at ISIMA
▷ *Address: Ensemble universitaire des Cézeaux, F-63172 Aubière, France.*
▷ *Supervisor : Pr. Jocelyn Sérot.*

Formation

2000-2004 PHD Thesis in Computer Vision and Robotic, Université Clermont 2 Blaise Pascal
Supervised by Pr. Jocelyn Sérot. Thesis title : *A Cluster for real time computer vision : architecture, tools and applications*, defended Dec. 1^{st} 2006
▷ *Comittee :*

- Jean-Thierry Lapresté, *professor université clermont 2*
- Daniel Etiemble, *professor université paris sud 11, referee*
- Frédéric Loulergue, *professor université d'Orléans, referee*
- Jocelyn Sérot, *professor université clermont 2, supervisor*
- Thierry Chateau, *assistant professor université clermont 2*
- Franck Cappello, *DR INRIA Saclay*

.

2002-2003 Master Thesis in Signal Processing and Computer Vision, Université Clermont 2 Blaise Pascal

▷ *Thesis: SIMD code generation on PowerPC using OCaml.*

2000-2003 Engineering Degree, Institut Supérieur d'Informatique, de Modélisation et de leurs Applications

---

Technical Skills

Programming	C++, C, x86 and PPC assembly, SSE, Altivec, NEON, MPI, OpenMP, Cuda, System programming, Shell
Systems	Linux / Unix, Mac OSX, Windows XP/7
Office	LaTeX, Microsoft Office

Languages

French	Native speaker
English	Read, write, speak fluently.

Research and Development

Specialization	Design and implementation of Parallel Programming Libraries ▷ *Domain Specific Languages.* ▷ *Generic Programming.* ▷ *Generative Programming.* ▷ *Parallel Programming Models.* ▷ *Algorithmic Skeletons.* ▷ *Domain Driven Engineering.* ▷ *Template meta-programming.*
Publications	Author or co-author of almost 40 publications in various journal and conferences ▷ *IEEE PACT, IEEE CiSE, Kluwer IJPP, ICCS, EuroPar, etc.*
Startup transfer	Scientific counseling in NumScale SAS

Teaching

2008-Now Assistant Professor at Polytech Paris Sud

- ▷ *Main courses : PHD and MSc level on programming, parallel computing, languages.*
- ▷ *Co-Supervisor of the practical teaching courses at Polytech from 2008 to 2011.*
- ▷ *Supervisor of the second year of engineering from 2012 to 2013.*
- ▷ *Supervisor of the System Programming courses in first year of engineering.*
- ▷ *Supervisor of the C++ course in first and second year of engineering.*
- ▷ *Supervisor of the parallel programming course in second year of engineering.*

2007-2008 Temporary teaching Assistant at Polytech Paris Sud

- ▷ *Main courses : MsC Level computer architecture.*

2007-2008 Temporary teaching Assistant at Ecole Polytechnique

- ▷ *High Performance Computing practical sessions.*

Thesis Supervisions

PHD Thesis

- ▷ Ian Masliah (M. Baboulin 50%, J. Falcou 50%), *Generic Programming for Linear Algebra Code Generation*, September 2013.
- ▷ Haixiong Ye (L. Lacassagne 40%, 30%, J. Falcou 30%), *Contributino to High-Level Synthesis Contribution : Impact of high-level algorithmic transforms*, January 2011 - Defense planned early 2014. Co-mentored zith ST MicroElectronics.
- ▷ Pierre Estérie (B. Rozoy 30%, J. Falcou 70%), *Generic and Generative Programming for High-Performance Computing*, September 2010 - Defense planned for early 2014. 6 co-authored publications.
- ▷ Khaled Hamidouche (D. Etiemble 30%, J. Falcou 70%), *Programming hierarchical heterogeneous parallel machines*, September 2008- November 2011, 6 co-authored publications. Now post-doctoral researcher at Ohio State University.

Engineer

- ▷ Eric Jourdanneau (full-time supervision), R&D engineer, October 2009 to August 2010.

Master Thesis

- ▷ Ian Masliah, March - August 2013. NT2 wrappers for MAGMA
- ▷ Gabriel Gallin, March - August 2013. NT2 back-end for OpenCL based FPGAs
- ▷ M2R Dimitrios Chasapis, November 2012 - July 2013. Future programming model implementation usign one-sided MPI primitives. Now a PHD student at Barcelona Supercomputing Center
- ▷ M2R Nicolas Lupinski, March - August 2011. COQ proof of parallel skeletons
- ▷ M2R Pierre Estérie, March - September 2010. Container and Algorithms for the CELL processor
- ▷ M2R Khaled Hamidouche, March - September 2008. MPI/OpenMP hybrid systems benchmarks

---

Research Projects

ITOC — Parallel implementation of non-intrusive X-ray based exploration
- ▷ *Partners: CEA (lead), LRI, LIX.*
- ▷ *Funding: 198Keuros - DIGITEO.*
- ▷ *Coordinator: (CEA).*
- ▷ *Provided CUDA/OpenCL based implementation of the main algorithms.*
- ▷ *Achieved speedup of 50 on the main reconstruction phase.*

MIDAS — Parallel implementation of cosmic background map reconstruction
- ▷ *Partners: LAC (lead), INRIA, LRI, CCC Berkeley.*
- ▷ *Funding: 500 Keuros - ANR.*
- ▷ *Coordinator: Radek Stompor (LAC).*
- ▷ *Lead the GPU based implementation effort.*

CELL-MPI — CELL based implementation of the MPI standard
- ▷ *Partners: LRI (lead).*
- ▷ *Funding: 100Keuros - DIGITEO OMTE (Technological transfer funding).*
- ▷ *Coordinator: Joel Falcou (LRI).*
- ▷ *Concluded by patent FR2967800.*

NT2 Transfer — Design and Development of a Matlab to C++ compiler
- ▷ *Partners: LRI (lead).*
- ▷ *Funding: 100Keuros - DIGITEO OMTE (Technological transfer funding).*
- ▷ *Coordinator: Joel Falcou (LRI).*
- ▷ *Concldued by the creation of NumScale SAS.*

Fluctus — Parallel implementation of a Coherent Lagrangian Structures simulator
- ▷ *Partners: LIMSI (lead), LRI.*
- ▷ *Funding: 205Keuros - DIGITEO.*
- ▷ *Coordinator: Luc Pastur (LIMSI).*
- ▷ *Provided CUDA/OpenCL based implementation of the main simulator.*
- ▷ *Achieved speedup of 250 on the main algorithm, allowing near real-time runs of experiments.*

Open Source Softwares

NT2 — C++ library providing an automatically parallelized MATLAB-like array language
- ▷ *Modular supports for new architectures.*
- ▷ *Available on* `http://www.github.com/NumScale/nt2`.

Cell-MPI — Implementation of the MPI standard on the CELL processor
- ▷ *Simplify writing CELL parallel applications.*
- ▷ *Patented technology (FR2967800).*

Quaff — C++ library for parallel skeleton based programming
- ▷ *Use meta-programming to derive task graph from parallel skeleton nesting.*
- ▷ *Currently discontinued in favor of NT2.*

BSP++ — C++ library implementing the BSP programming model as a generic component
- ▷ *Provide supports for nested MPI, OpenMP and CELL architectures.*
- ▷ *Available at* `http://github.com/jfalcou/bsppp`.

Boost.SIMD — C++ standard compliant library for portable SIMD programming
- ▷ *Simplify SIMD programming by providing standard compatible constructs.*
- ▷ *Supports SSEx, AVX1-2, NEON, Altivec.*
- ▷ *Available on* `http://www.github.com/NumScale/nt2`.

Scientific Activities

Expertise	Member of the French national body for C/C++ ISO normalization (JTC1/SC22/WG21) since 2012 ▷ *High-performance computing expert.*
Administration	Co-leader of the PARSYS team at LRI since 2013 ▷ *Member of the Laboratory Council since 2013.* ▷ *Member of the Laboratory Technology Transfer Council since 2011.*
Communities	▷ Program Committee Member for international conferences ▷ Program Co-chair of C++NOW évolution `http://cppnow.org/` ▷ Co-chair of the Workshop on Programming Models for SIMD/Vector Processing `https://sites.google.com/site/wpmvp2014/home` ▷ Member of the CNRS Research Group on Software Design (GDR GPL) ▷ Member of the CNRS Research Group on High-Level Languages and Models for parallel programming (LaMHA). ▷ Reviewers for more than 10 international journal and conference (PARCO, PACT, JPDC, IPDPS, EuroPar)
Teaching	▷ Head of the Computer Science Engineering Practical Track at Polytech Paris Sud from 2008 to 2010 ▷ Coordinator of the 4th year of Computer Science Engineering Track at Polytech Paris Sud from 2011 to 2012

Other Activities

Sports	Mountain biking
Hobbies	Collectible comics and memorabilia, foreign movies
Associations	Founding member of the french C++ *User Group* (http://www.meetup.com/User-Group-Cpp-Francophone/)

Bibliography

— PHD Thesis —

[1] Joel Falcou. *Un Cluster pour la Vision Temps Réel – Architecture, Outils et Applications.* PhD thesis, University Clermont II, Blaise Pascal, France, December 2006.

— Patents —

[2] Joel Falcou, Lionel Lacassagne, Sebastian Schaetz, and Claude Tadonki. Procédé de synchronisation et de transfert de données entre des procresseurs reliés par des canaux dma. In *Patent number FR2967800*, June 2012.

— International Journal Papers —

[3] Khaled Hamidouche, Fernando Machado Mendonca, Joel Falcou, Alba Cristina Magalhaes Alves de Melo, and Daniel Etiemble. Parallel smith-waterman comparison on multicore and manycore computing platforms with bsp++. *International Journal of Parallel Programming*, 41(1):111–136, 2013.

[4] Mikolaj Szydlarski, Pierre Esterie, Joel Falcou, Laura Grigori, and Radek Stompor. Parallel spherical harmonic transforms on heterogeneous architectures (graphics processing units/multi-core cpus). *Concurrency and Computation: Practice and Experience*, 2013.

[5] Pierre Esterie, Mathias Gaunard, Joel Falcou, et al. Exploiting multimedia extensions in c++: A portable approach. *Computing in Science & Engineering*, 14(5):72–77, 2012.

[6] Tarik Saidani, Lionel Lacassagne, Joel Falcou, Claude Tadonki, and Samir Bouaziz. Parallelization schemes for memory optimization on the cell processor: A case study on the harris corner detector. In *Transactions on high-performance embedded architectures and compilers III*, pages 177–200. Springer Berlin Heidelberg, 2011.

[7] Joel Falcou. Parallel programming with skeletons. *Computing in Science & Engineering*, 11(3): 58–63, 2009.

[8] Tarik Saidani, Joel Falcou, Lionel Lacassagne, and Samir Bouaziz. Altivec vector unit customization for embedded systems. *IJCSA*, 5(3a):20–32, 2008.

[9] Joel Falcou, Jocelyn Sérot, Thierry Chateau, and Jean-Thierry Lapresté. Quaff: efficient c++ design for parallel skeletons. *Parallel Computing*, 32(7):604–615, 2006.

[10] Joel Falcou and Jocelyn Serot. Eve, an object oriented simd library. *Scalable Computing: Practice and Experience*, 6(4), 2001.

— National Journal Papers —

[11] Joel Falcou and Jocelyn Sérot. Une bibliotheque métaprogrammée pour la programmation parallele. *TSI. Technique et science informatiques*, 28(5):645–675, 2009.

— Peer-reviewed International Conference Papers —

[12] Yushan Wang, Marc Baboulin, Jack Dongarra, Joël Falcou, Yann Fraigneau, and Olivier Le Maître. A parallel solver for incompressible fluid flows. *Procedia Computer Science*, 18:439–448, 2013.

[13] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. Boost. simd: generic programming for portable simdization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 431–432. ACM, 2012.

[14] H Ye, Lionel Lacassagne, Daniel Etiemble, L Cabaret, Joel Falcou, Andrés Romero, and O Florent. Impact of high level transforms on high level synthesis for motion detection algorithm. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pages 1–8. IEEE, 2012.

[15] Giulio Fabbian, M Szydlarski, R Stompor, L Grigori, and J Falcou. Spherical harmonic transforms with s2hat (scalable spherical harmonic transform) library. In *Astronomical Data Analysis Software and Systems XXI*, volume 461, pages 61–66, 2012.

[16] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. A framework for an automatic hybrid mpi+ openmp code generation. In *Proceedings of the 19th High Performance Computing Symposia*, pages 48–55. Society for Computer Simulation International, 2011.

[17] Khaled Hamidouche, Fernando Machado Mendonca, Joel Falcou, and Daniel Etiemble. Parallel biological sequence comparison on heterogeneous high performance computing platforms with bsp++. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pages 136–143. IEEE, 2011.

[18] Tarik Saidani, Joel Falcou, Claude Tadonki, Lionel Lacassagne, and Daniel Etiemble. Algorithmic skeletons within an embedded domain specific language for the cell processor. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 67–76. IEEE, 2009.

[19] Joel Falcou. High level parallel programming edsl-a boost libraries use case. In *BOOSTâĂŹCON 2009*, volume 9, 2009.

[20] Joel Falcou, Jocelyn Sérot, Lucien Pech, and Jean-Thierry Lapresté. Meta-programming applied to automatic smp parallelization of linear algebra code. In *Euro-Par 2008–Parallel Processing*, pages 729–738. Springer Berlin Heidelberg, 2008.

[21] Jocelyn Serot and Joel Falcou. Functional meta-programming for parallel skeletons. In *Computational Science–ICCS 2008*, pages 154–163. Springer Berlin Heidelberg, 2008.

[22] Joel Falcou and Jocelyn Serot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. *Parallel Computing: Architectures, Algorithms and Applications (Proc. of PARCO 2007, Julich, Germany)*, 38:243–252, 2008.

[23] Joel Falcou, Jocelyn Sérot, Thierry Chateau, Frédéric Jurie, et al. A parallel implementation of a 3d reconstruction algorithm for real-time vision. In *PARCO*, pages 663–670, 2005.

[24] Joel Falcou and Jocelyn Sérot. Eve, an object oriented simd library. In *Computational Science-ICCS 2004*, pages 314–321. Springer Berlin Heidelberg, 2004.

[25] Joel Falcou and Jocelyn Serot. Application of template-based metaprogramming compilation techniques to the efficient implementation of image processing algorithms on simd-capable processors. *ACIVS 2004*, 2004.

— Peer-reviewed National Conference Papers —

[26] Joel Falcou, Tarik Saidani, Lionel Lacassagne, and Daniel Etiemble. Programmation par squelettes algorithmiques pour le processeur cell. In *SYMPA 2008*, 2008.

[27] Jocelyn Sérot, Joël Falcou, et al. Métaprogrammation fonctionnelle appliquée à la génération d'un dsl dédié à la programmation parallèle. In *JFLA (Journées Francophones des Langages Applicatifs)*, pages 153–171, 2008.

[28] Joel Falcou, Jocelyn Sérot, Thierry Chateau, and Frédéric Jurie. Un cluster de calcul hybride pour les applications de vision temps réel. *20Âř Colloque sur le traitement du signal et des images, FRA, 2005*, 2005.

[29] Joel Falcou and Jocelyn Sérot. Camlg4: une bibliothèque de calcul parallèle pour objective caml. In *JFLA*, pages 139–152, 2003.

— Peer-reviewed International Workshops —

[30] H Ye, L Lacassagne, J Falcou, D Etiemble, L Cabaret, and O Florent. High level tranforms toreduce energy consumption of signal and image processing operators. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013 23rd International Workshop on*, pages 247–254. IEEE, 2013.

[31] Lionel DAMEZ, Loic SIELER, Joel FALCOU, and Jean Pierre DERUTIN. Méthode d'implantation parallèle d'applications de tsi sur sopc. *XXIIe colloque GRETSI (traitement du signal et des images), Dijon (FRA), 8-11 septembre 2009*, 2009.

[32] Joel Falcou, Jean-Thierry Lapresté, Thierry Chateau, Jocelin Sérot, et al. Nt2: Une bibliothèque haute-performance pour la vision artificielle. *ORASIS 2007-Orasis 2007, Congrès Jeunes Chercheurs en Vision par Ordinateur*, 2007.

[33] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. Hybrid bulk synchronous parallelism library for clustered smp architectures. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 55–62. ACM, 2010.

[34] Claude Tadonki, Lionel Lacassagne, Tarik Saidani, Joel Falcou, and Khaled Hamidouche. The harris algorithm revisited on the cell processor. In *International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies 2010*, 2010.

[35] Khaled Hamidouche, Alexandre Borghi, Pierre Esterie, Joel Falcou, and Sylvain Peyronnet. Three high performance architectures in the parallel apmc boat. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 20–27. IEEE, 2010.

[36] Joel Falcou, Thierry Chateau, Jocelin Sérot, Jean-Thierry Lapresté, et al. Real time parallel implementation of a particle filter based visual tracking. In *CIMCV 2006-Workshop on Computation Intensive Methods for Computer Vision at ECCV 2006*, 2006.

— Technical Reports —

[37] Ioan Ovidiu, Laura Grigori, Joel Falcou, and Radek Stompor. Spherical harmonic transform with gpus. 2010.

[38] F Guniat, L Pastur, F Lusseyran, J Falcou, M Ammi, and Y Fraigneau. Fast identification of lagrangian coherent structures-bifd2011.

[39] Mikolaj Szydlarski, Pierre Esterie, Joel Falcou, Laura Grigori, and Radek Stompor. Parallel spherical harmonic transforms on heterogeneous architectures (gpus/multi-core cpus). *arXiv preprint arXiv:1106.0159*, 2011.

Software Abstractions for Parallel Hardware Architectures

Abstract: Performing large, intensive or non-trivial computing on array like data structures is one of the most common task in scientific computing, video game development and other fields. This matter of fact is backed up by the large number of tools, languages and libraries to perform such tasks. If we restrict ourselves to C++ based solutions, more than a dozen such libraries exists from BLAS/LAPACK C++ binding to template meta-programming based Blitz++ or Eigen.

If all of these libraries provide good performance or good abstraction, none of them seems to fit the need of so many different user types. Moreover, as parallel system complexity grows, the need to maintain all those components quickly become unwieldy. This thesis explores various software design techniques - like Generative Programming, MetaProgramming and Generic Programming - and their application to the implementation of various parallel computing libraries in such a way that abstraction and expressiveness are maximized while efficiency overhead is minimized.

Keywords: parallel programming, parallel skeletons, generic programming, generative programming, meta-programming.

Abstractions Logicielles pour Architectures Paralllèles

Résumé: Un nombre croissant de domaines applicatifs allant du calcul scientifique au jeu vidéo reposent sur la manipulation efficace et non triviale de structure de données de type tableau. Cet état de fait est d'autant plus criant au vu du nombre d'outils libres ou propriétaires visant á simplifier le développement de telles applications. En se limitant á C++ , plus d'une douzaine de telles bibliothèques sont disponibles comme Blitz ou Eigen.

Si toutes ces bibliothèques fournissent ou un bon niveau d'abstraction ou des performances élevées, il leur est difficile d'accommoder un tel spectre d'utilisateurs. De plus, devant la multiplication et la complexification des systèmes de calculs parallèles, il devient difficile de maintenir et de développer de nouvelles fonctionalistes tirant parties de ce matériel.

Cette thèse explore comment des techniques de développement logiciel comme la programmation generative, la programmation générique et la méta-programmation peuvent être utilisées afin de proposer une nouvelle méthode de développement adaptées á ce type de contraintes et permettant de concilier haut niveau d'abstraction et haut niveau de performance.

Mots Clés: programmation parallèle, squelettes parallèles, programmation générique, programmation générative, méta-programmation.

Bibliography

[Abrahams 2004] David Abrahams and Aleksey Gurtovoy. C++ template metaprogramming: Concepts, tools, and techniques from boost and beyond (c++ in depth series). Addison-Wesley Professional, 2004. (Cited on page 23.)

[Aggarwal 1989] A. Aggarwal, A. K. Chandra and M. Snir. *On Communication Latency in PRAM Computations.* In Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '89, pages 11–21, New York, NY, USA, 1989. ACM. (Cited on page 8.)

[Aggarwal 1990] Alok Aggarwal, Ashok K. Chandra and Marc Snir. *Communication complexity of {PRAMs}.* Theoretical Computer Science, vol. 71, no. 1, pages 3 – 28, 1990. (Cited on page 9.)

[AMD] AMD. *AMD Core Math Library.* `http://developer.amd.com/libraries/acml/pages/default.aspx`. (Cited on page 58.)

[Aneja 2009] Kanur Aneja, Florence Laguzet, Lionel Lacassagne and Alain Merigot. *Video-rate image segmentation by means of region splitting and merging.* Proc. Signal and Image Processing Applications, pages 437–442, 2009. (Cited on page 80.)

[Ayguadé 2009] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan and Guansong Zhang. *The design of openmp tasks.* Parallel and Distributed Systems, IEEE Transactions on, vol. 20, no. 3, pages 404–418, 2009. (Cited on page 74.)

[Bacci 1995] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti and M. Vanneschi. *P3L: A Structured High Level Programming Language And Its Structured Support.* Concurrency: Practice and Experience, pages 225–255, 1995. (Cited on page 15.)

[Baker Jr 1977] Henry C Baker Jr and Carl Hewitt. *The incremental garbage collection of processes.* In ACM SIGART Bulletin, volume 12, pages 55–59. ACM, 1977. (Cited on page 74.)

[Beckman 2006] P. Beckman, K. Iskra, K. Yoshii and S. Coghlan. *The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale.* In Cluster Computing, 2006 IEEE International Conference on, pages 1–12, 2006. (Cited on page 16.)

[Benoit 2005a] Anne Benoit and Murray Cole. *Two Fundamental Concepts in Skeletal Parallel Programming.* In Proceedings of the 5th International Conference on Computational Science - Volume Part II, ICCS'05, pages 764–771, Berlin, Heidelberg, 2005. Springer-Verlag. (Cited on page 15.)

[Benoit 2005b] Anne Benoit, Murray Cole, Stephen Gilmore and Jane Hillston. *Flexible Skeletal Programming with Eskel.* In Proceedings of the 11th International Euro-Par Conference on Parallel Processing, Euro-Par'05, pages 761–770, Berlin, Heidelberg, 2005. Springer-Verlag. (Cited on page 15.)

[Benoit 2008] Anne Benoit and Yves Robert. *Mapping Pipeline Skeletons Onto Heterogeneous Platforms.* J. Parallel Distrib. Comput., vol. 68, no. 6, pages 790–808, June 2008. (Cited on page 15.)

[Beran 1999] Martin Beran. *Decomposable Bulk Synchronous Parallel Computers.* In Jan Pavelka, Gerard Tel and Miroslav Bartosek, editeurs, SOFSEM'99: Theory and Practice of Informatics, volume 1725 of *Lecture Notes in Computer Science*, pages 349–359. Springer Berlin Heidelberg, 1999. (Cited on page 32.)

[Bikshandi 2006] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua and Christoph von Praun. *Programming for Parallelism and Locality with Hierarchically Tiled Arrays.* In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM. (Cited on page 11.)

[Blagojević 2010] Filip Blagojević, Paul Hargrove, Costin Iancu and Katherine Yelick. *Hybrid PGAS Runtime Support for Multicore Nodes.* In Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS '10, pages 3:1–3:10, New York, NY, USA, 2010. ACM. (Cited on page 12.)

[Blelloch 1997] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias and Marco Zagha. *Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors.* IEEE Trans. Parallel Distrib. Syst., vol. 8, no. 9, pages 943–958, 1997. (Cited on page 11.)

[Bonorden 1999] Olaf Bonorden, Ben H. H. Juurlink, Ingo von Otte and Ingo Rieping. *The Paderborn University BSP (PUB) Library - Design, Implementation and Performance.* In IPPS/SPDP, pages 99–104, 1999. (Cited on page 11.)

[Borghi 2008] Alexandre Borghi, Thomas Hérault, Richard Lassaigne and Sylvain Peyronnet. *Cell Assisted APMC.* In QEST, pages 75–76, 2008. (Cited on page 36.)

[Boukerche 2009] Azzedine Boukerche, Rodolfo Bezerra Batista and Alba Cristina Magalhaes Alves de Melo. *Exact pairwise alignment of megabase genome biological sequences using a novel z-align parallel strategy.* Parallel and Distributed Processing Symposium, International, vol. 0, pages 1–8, 2009. (Cited on page 45.)

[Bright 2014] Walter Bright. *Templates Revisited*. http://dlang.org/templates-revisited.html, 2014. (Cited on page 23.)

[Brodman 2008] James Brodman, BasilioB. Fraguela, MarìaJ. Garzaran and David Padua. *Design Issues in Parallel Array Languages for Shared Memory*. In Mladen Berekovic, Nikitas Dimopoulos and Stephan Wong, editeurs, Embedded Computer Systems: Architectures, Modeling, and Simulation, volume 5114 of *Lecture Notes in Computer Science*, pages 208–217. Springer Berlin Heidelberg, 2008. (Cited on page 11.)

[Brown 2011] Kevin J Brown, Arvind K Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky and Kunle Olukotun. *A heterogeneous parallel framework for domain-specific languages*. In Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on, pages 89–100. IEEE, 2011. (Cited on page 29.)

[Buss 2010] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato and Lawrence Rauchwerger. *STAPL: Standard Template Adaptive Parallel Library*. In Proceedings of the 3rd Annual Haifa Experimental Systems Conference, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM. (Cited on page 22.)

[Cappello 2010] Franck Cappello. *Grid 5000*, 2010. (Cited on pages 41 and 88.)

[Cehn 2003] C. Cehn and B. Schmidt. *Computing large-scale alignments on a multi-cluster*. In Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on, pages 38–45, Dec 2003. (Cited on page 45.)

[Chen 2014] C.L. Philip Chen and Chun-Yang Zhang. *Data-intensive applications, challenges, techniques and technologies: A survey on Big Data*. Information Sciences, vol. 275, no. 0, pages 314 – 347, 2014. (Cited on page 4.)

[Ciarpaglini 1997] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi and S. Pelagatti. *ANACLETO: A Template-based P3L Compiler*. Proceedings of the PCW'97, 1997. (Cited on page 15.)

[Ciechanowicz 2009] Philipp Ciechanowicz, Michael Poldner and Herbert Kuchen. *The Munster Skeleton Library Muesli: A comprehensive overview*. ERCIS Working Papers 7, Westfalsche Wilhelms-Universitat Munster (WWU) - European Research Center for Information Systems (ERCIS), 2009. (Cited on page 16.)

[Ciechanowicz 2010] Philipp Ciechanowicz and Herbert Kuchen. *Enhancing Muesli's data parallel skeletons for multi-core computer architectures*. In High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on, pages 108–113. IEEE, 2010. (Cited on page 73.)

[Cole 1989] Murray Cole. Algorithmic skeletons: structured management of parallel computation. MIT Press, 1989. (Cited on page 13.)

[Cole 2004] Murray Cole. *Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming.* Parallel Computing, vol. 30, no. 3, pages 389–406, March 2004. (Cited on pages 14, 15 and 73.)

[Committee 2011] ISO JTC1/SC22/WG21 C++ Standards Committee. *C++ standard library:Threads*, 2011. (Cited on page 18.)

[Courcoubetis 1995] Costas Courcoubetis and Mihalis Yannakakis. *The Complexity of Probabilistic Verification.* J. ACM, vol. 42, no. 4, pages 857–907, July 1995. (Cited on page 36.)

[Culler 1993] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian and Thorsten von Eicken. *LogP: Towards a Realistic Model of Parallel Computation.* SIGPLAN Not., vol. 28, no. 7, pages 1–12, 1993. (Cited on page 9.)

[Czarnecki 1998] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde and Todd L. Veldhuizen. *Generative Programming and Active Libraries.* In Generic Programming, pages 25–39, 1998. (Cited on page 47.)

[Czarnecki 2000] Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative programming - methods, tools and applications. Addison-Wesley, 2000. (Cited on page 48.)

[de Guzman] Joel de Guzman, Dan Marsden and Tobias Schwinger. *Boost.Fusion Library.* `http://www.boost.org/doc/libs/release/libs/fusion/doc/html`. (Cited on page 62.)

[Demaille 2006] Akim Demaille, Thomas Hérault and Sylvain Peyronnet. *Probabilistic verification of sensor networks.* In RIVF, pages 45–54, 2006. (Cited on page 36.)

[Duran 2012] Alejandro Duran and Michael Klemm. *The Intel® Many Integrated Core Architecture.* In High Performance Computing and Simulation (HPCS), 2012 International Conference on, pages 365–366. IEEE, 2012. (Cited on page 56.)

[El-Ghazawi 2003] Tarek El-Ghazawi, William Carlson, Thomas Sterling and Katherine Yelick. Upc: Distributed shared-memory programming. Wiley-Interscience, 2003. (Cited on page 12.)

[Emoto 2007] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi and Masato Takeichi. *A Compositional Framework for Developing Parallel Programs on Two-dimensional Arrays.* Int. J. Parallel Program., vol. 35, no. 6, pages 615–658, December 2007. (Cited on page 16.)

[Esterie 2012] Pierre Esterie, Mathias Gaunard, Joel Falcou and Jean-Thierry Lapresté. *Exploiting Multimedia Extensions in C++: A Portable Approach.* Computing in Science & Engineering, vol. 14, no. 5, pages 72–77, 2012. (Cited on page 71.)

[Estérie 2013] Pierre Estérie, Mathias Gaunard and Joel Falcou. *N3561 - A proposal to add single instruction multiple data computation to the standard library.* JTC1/SC22/WG21 - The C++ Standards Committee, 2013. (Cited on page 55.)

[Esterie 2014a] Pierre Esterie. *Multi-architectural Support: A Generic and Generative Approach.* PhD thesis, University Paris Sud, 2014. (Cited on pages 55 and 68.)

[Estérie 2014b] Pierre Estérie, Joel Falcou, Mathias Gaunard and Jean-Thierry Lapresté. *Boost.SIMD: generic programming for portable SIMDization.* In Proceedings of the 2014 Workshop on Workshop on programming models for SIMD/Vector processing, pages 1–8. ACM, 2014. (Cited on pages 55, 68 and 77.)

[Esterie 2014c] Pierre Esterie, Joel Falcou, Mathias Gaunard, Jean-Thierry Lapresté and Lionel Lacassagne. *The numerical template toolbox: A modern C++ design for scientific computing.* Journal of Parallel and Distributed Computing, 2014. (Cited on page 69.)

[Fahmy 1996] Amr Fahmy and Abdelsalam Heddaya. *Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk*, 1996. (Cited on page 11.)

[Fortune 1978] Steven Fortune and James Wyllie. *Parallelism in Random Access Machines.* In Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM. (Cited on page 8.)

[Fox 2011] Thomas Fox, Michael Gschwind and Jaime Moreno. *QPX Architecture: Quad Processing eXtension to the Power ISA.* Software: Practice and Experience, 2011. (Cited on page 56.)

[Friedman 1976] Daniel P Friedman and David Stephen Wise. The impact of applicative programming on multiprocessing. Indiana University, Computer Science Department, 1976. (Cited on page 74.)

[Gibbons 1989] P. B. Gibbons. *A More Practical PRAM Model.* In Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '89, pages 158–168, New York, NY, USA, 1989. ACM. (Cited on page 9.)

[Gokhale 2003] Aniruddha S. Gokhale, Douglas C. Schmidt, Tao Lu, Balachandran Natarajan and Nanbor Wang. *CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Applications.* In Middleware Workshops, pages 300–306, 2003. (Cited on page 49.)

[Gonzalez-Velez 2010] Horacio Gonzalez-Velez and Mario Leyton. *A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers.* Softw. Pract. Exper., vol. 40, no. 12, pages 1135–1160, November 2010. (Cited on page 15.)

[Gonzalez 2000] Jesus A. Gonzalez, Coromoto Leon, Fabiana Piccoli, Marcela Printista, José' L. Roda, Casiano Rodriguez and Francisco de Sande. *Oblivious BSP (Research Note).* In Arndt Bode, Thomas Ludwig 0002, Wolfgang Karl and Roland Wismuller, editeurs, Euro-Par, volume 1900 of *Lecture Notes in Computer Science*, pages 682–685. Springer, 2000. (Cited on page 11.)

[Goudreau 1999] Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel and Thanasis Tsantilas. *Portable and Efficient Parallel Computing Using the BSP Model.* IEEE Trans. Comput., vol. 48, no. 7, pages 670–689, July 1999. (Cited on page 11.)

[Gregor 2006] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Bjarne Stroustrup, Gabriel Dos Reis and Andrew Lumsdaine. *Concepts: linguistic support for generic programming in C++.* In OOPSLA, pages 291–310, 2006. (Cited on page 20.)

[Guelton 2014] Serge Guelton, Joël Falcou and Pierrick Brunet. *Exploring the vectorization of python constructs using pythran and boost SIMD.* In Proceedings of the 2014 Workshop on Workshop on programming models for SIMD/Vector processing, pages 79–86. ACM, 2014. (Cited on page 80.)

[Haeri 2012] Seyed Hossein Haeri, Sibylle Schupp and Jonathan Hüser. *Using Functional Languages to Facilitate C++ Metaprogramming.* In Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming, WGP '12, pages 33–44, New York, NY, USA, 2012. ACM. (Cited on page 23.)

[Hamidouche 2010a] Khaled Hamidouche, Alexandre Borghi, Pierre Esterie, Joel Falcou and Sylvain Peyronnet. *Three high performance architectures in the parallel APMC boat.* In Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on, pages 20–27. IEEE, 2010. (Cited on page 31.)

[Hamidouche 2010b] Khaled Hamidouche, Joel Falcou and Daniel Etiemble. *Hybrid bulk synchronous parallelism library for clustered SMP architectures.* In Proceedings of the fourth international workshop on High-level parallel programming and applications, pages 55–62. ACM, 2010. (Cited on pages 31, 32 and 41.)

[Hamidouche 2011a] Khaled Hamidouche. *Programmation des machines Hiérarchiques et Hétérogènes*. PhD thesis, University Paris Sud, 2011. (Cited on page 31.)

[Hamidouche 2011b] Khaled Hamidouche, Joel Falcou and Daniel Etiemble. *A framework for an automatic hybrid MPI+openMP code generation*. In Proceedings of the 19th High Performance Computing Symposia, pages 48–55. Society for Computer Simulation International, 2011. (Cited on page 31.)

[Hamidouche 2011c] Khaled Hamidouche, Fernando Machado Mendonca, Joel Falcou and Daniel Etiemble. *Parallel biological sequence comparison on heterogeneous high performance computing platforms with BSP++*. In Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on, pages 136–143. IEEE, 2011. (Cited on page 31.)

[Haney 1999] Scott Haney and James Crotinger. *Pete: The portable expression template engine*. Dr. Dobb's journal, vol. 24, no. 10, 1999. (Cited on page 24.)

[Hérault 2004] Thomas Hérault, Richard Lassaigne, Frédéric Magniette and Sylvain Peyronnet. *Approximate Probabilistic Model Checking*. In VMCAI, pages 73–84, 2004. (Cited on page 36.)

[Hérault 2006] Thomas Hérault, Richard Lassaigne and Sylvain Peyronnet. *APMC 3.0: Approximate Verification of Discrete and Continuous Time Markov Chains*. In QEST, pages 129–130, 2006. (Cited on pages 36 and 37.)

[Hill 1998] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin J. Lang, Satish Rao, Torsten Suel, Thanasis Tsantilas and Rob H. Bisseling. *BSPlib: The BSP programming library*. Parallel Computing, vol. 24, no. 14, pages 1947–1980, 1998. (Cited on pages 10 and 11.)

[Hinton 2006] Andrew Hinton, Marta Kwiatkowska, Gethin Norman and David Parker. *PRISM: A Tool for Automatic Verification of Probabilistic Systems*. In Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'06, pages 441–444, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on pages 36 and 37.)

[Intel] Intel. *Math Kernel Library*. `http://developer.intel.com/software/products/mkl/`. (Cited on page 58.)

[Jang 2011] Minwoo Jang, Kukhyun Kim and Kanghee Kim. *The performance analysis of ARM NEON technology for mobile platforms*. In Proceedings of the 2011 ACM Symposium on Research in Applied Computation, RACS '11, pages 104–106, New York, NY, USA, 2011. ACM. (Cited on page 56.)

[Jarvi 1998] Jaakko Jarvi. *Compile time recursive objects in C++*. In Technology of Object-Oriented Languages, 1998. TOOLS 27. Proceedings, pages 66–77. IEEE, 1998. (Cited on page 23.)

[Kaiser 2009] Hartmut Kaiser, Maciej Brodowicz and Thomas Sterling. *Parallex an advanced parallel execution model for scaling-impaired applications.* In Parallel Processing Workshops, 2009. ICPPW'09. International Conference on, pages 394–401. IEEE, 2009. (Cited on page 74.)

[Karp 1983] Richard M. Karp and Michael Luby. *Monte-Carlo Algorithms for Enumeration and Reliability Problems.* In Proceedings of the 24th Annual Symposium on Foundations of Computer Science, SFCS '83, pages 56–64, Washington, DC, USA, 1983. IEEE Computer Society. (Cited on page 36.)

[Kretz 2012] Matthias Kretz and Volker Lindenstruth. *Vc: A C++ library for explicit vectorization.* Software: Practice and Experience, vol. 42, no. 11, pages 1409–1430, 2012. (Cited on page 58.)

[Kuchen 2002] Herbert Kuchen. A skeleton library. Springer, 2002. (Cited on page 73.)

[Kurzak 2009] Jakub Kurzak, Wesley Alvaro and Jack Dongarra. *Optimizing matrix multiplication for a short-vector SIMD architecture : CELL processor.* Parallel Computing, vol. 35, no. 3, pages 138 – 150, 2009. (Cited on page 56.)

[Lacassagne 2009] Lionel Lacassagne, Antoine Manzanera, Julien Denoulet and Alain Mérigot. *High performance motion detection: some trends toward new embedded architectures for vision systems.* Journal of Real-Time Image Processing, vol. 4, pages 127–146, 2009. (Cited on pages 66 and 67.)

[Lacassagne 2011] Lionel Lacassagne and Bertrand Zavidovique. *Light speed labeling: efficient connected component labeling on RISC architectures.* Journal of Real-Time Image Processing, vol. 6, no. 2, pages 117–135, 2011. (Cited on page 80.)

[Leißa 2014] Roland Leißa, Immanuel Haffner and Sebastian Hack. *Sierra: A SIMD Extension for C++.* In Proceedings of the 1st International Workshop on Programming Models for SIMD/Vector Processing, WPMVP '14, 2014. (Cited on page 58.)

[Li 2012] Chong Li and Gaétan Hains. *SGL: Towards a Bridging Model for Heterogeneous Hierarchical Platforms.* Int. J. High Perform. Comput. Netw., vol. 7, no. 2, pages 139–151, 2012. (Cited on page 11.)

[Loulergue 2002] F. Loulergue. *Implementation of a Functional Bulk Synchronous Parallel Programming Library.* In 14^{extth} IASTED International Conference on Parallel and Distributed Computing Systems, pages 452–457, Cambridge, USA, November 2002. ACTA Press. (Cited on page 11.)

[Marques 2013] Ricardo Marques, Hervé Paulino, Fernando Alexandre and Pedro D. Medeiros. *Algorithmic Skeleton Framework for the Orchestration of GPU Computations.* In Proceedings of the 19th International Conference on

Parallel Processing, Euro-Par'13, pages 874–885, Berlin, Heidelberg, 2013. Springer-Verlag. (Cited on page 16.)

[Matsuzaki 2004] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu and Yoshiki Akashi. *A Fusion-Embedded Skeleton Library.* In Marco Danelutto, Marco Vanneschi and Domenico Laforenza, editeurs, Euro-Par 2004 Parallel Processing, volume 3149 of *Lecture Notes in Computer Science*, pages 644–653. Springer Berlin Heidelberg, 2004. (Cited on page 16.)

[Matsuzaki 2006] Kiminori Matsuzaki, Zhenjiang Hu and Masato Takeichi. *Parallel Skeletons for Manipulating General Trees.* Parallel Comput., vol. 32, no. 7, pages 590–603, September 2006. (Cited on page 16.)

[McCutchan 2014] John McCutchan, Haitao Feng, Nicholas Matsakis, Zachary Anderson and Peter Jensen. *A SIMD Programming Model for Dart, Javascript,and Other Dynamically Typed Scripting Languages.* In Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing, WPMVP '14, pages 71–78, New York, NY, USA, 2014. ACM. (Cited on page 80.)

[Moreau 2003] Pierre-Etienne Moreau, Christophe Ringeissen and Marian Vittek. *A pattern matching compiler for multiple target languages.* In Proceedings of the 12th international conference on Compiler construction, CC'03, pages 61–76, Berlin, Heidelberg, 2003. Springer-Verlag. (Cited on page 29.)

[Mount 2004] David W. Mount. Bioinformatics - sequence and genome analysis. Cold Spring Harbor Laboratory Press, 2004. (Cited on page 39.)

[Niebler 2007] Eric Niebler. *Proto : A compiler construction toolkit for DSELs.* In Proceedings of ACM SIGPLAN Symposium on Library-Centric Software Design, 2007. (Cited on pages 25 and 73.)

[Noorian 2009] Mahdi Noorian, Hamidreza Pooshfam, Zeinab Noorian and Rosni Abdullah. *Performance Enhancement of Smith-Waterman Algorithm Using Hybrid Model: Comparing the MPI and Hybrid Programming Paradigm on SMP Clusters.* In SMC, pages 492–497. IEEE, 2009. (Cited on page 45.)

[Numrich 1998] Robert W. Numrich and John Reid. *Co-array Fortran for Parallel Programming.* SIGPLAN Fortran Forum, vol. 17, no. 2, pages 1–31, August 1998. (Cited on page 12.)

[Nuzman 2006] Dorit Nuzman and Richard Henderson. *Multi-platform auto-vectorization.* In Proceedings of the International Symposium on Code Generation and Optimization, pages 281–294. IEEE Computer Society, 2006. (Cited on page 57.)

[Nuzman 2011] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen and Ayal Zaks. *Vapor SIMD:*

Auto-vectorize once, run everywhere. In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pages 151–160. IEEE Computer Society, 2011. (Cited on page 58.)

[OpenMP Architecture Review Board 2013] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.0*, 2013. (Cited on page 74.)

[Parent 2014] Sean Parent, Mat Marcus and Foster Brereton. *Adobe Generic Image Library*, 2014. (Cited on page 21.)

[Pfister 1995] Gregory F. Pfister. In search of clusters: The coming battle in lowly parallel computing. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995. (Cited on page 41.)

[Pharr 2012] Matt Pharr and William R Mark. *ispc: A SPMD compiler for high-performance CPU programming*. In Innovative Parallel Computing (InPar), 2012, pages 1–13. IEEE, 2012. (Cited on page 58.)

[Philipp 2012] Haller Philipp, Prokopec Aleksandar, Miller Heather, Klang Viktor, Kuhn Roland and Jovanovic Vojin. *Scala Documentation: Futures and Promises*, 2012. (Cited on page 17.)

[PixelGlow Software 2005] PixelGlow Software. *The MacSTL Library*, 2005. (Cited on page 58.)

[Pnueli 1986] Amir Pnueli and Lenore Zuck. *Verification of multiprocess probabilistic protocols*. Distributed Computing, vol. 1, no. 1, pages 53–72, 1986. (Cited on page 38.)

[Poldner 2005] Michael Poldner and Herbert Kuchen. *Scalable Farms*. In PARCO, pages 795–802, 2005. (Cited on page 14.)

[Rajko 2004] Stjepan Rajko and Srinivas Aluru. *Space and Time Optimal Parallel Sequence Alignments*. IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 12, pages 1070–1081, 2004. (Cited on page 45.)

[Reinders 2010] James Reinders. Intel threading building blocks: outfitting c++ for multi-core processor parallelism. O'Reilly Media, Inc., 2010. (Cited on page 74.)

[Reynders 1996] John VW Reynders, Paul J Hinker, Julian C Cummings, Susan R Atlas, Subhankar Banerjee, William F Humphrey, Steve R Karmesin, Katarzyna Keahey, Marikani Srikant and Mary Dell Tholburn. *POOMA: A framework for scientific simulations on parallel architectures*. Parallel Programming in C+, pages 547–588, 1996. (Cited on page 24.)

[Robison 2013] Arch D Robison. *Composable Parallel Patterns with Intel Cilk Plus*. Computing in Science & Engineering, vol. 15, no. 2, pages 0066–71, 2013. (Cited on page 58.)

[Russell 2011] Francis P. Russell, Michael R. Mellor, Paul H.J. Kelly and Olav Beckmann. *DESOLA: An active linear algebra library using delayed evaluation and runtime code generation*. Science of Computer Programming, vol. 76, no. 4, pages 227 – 242, 2011. (Cited on page 29.)

[Schaller 1997] Robert R. Schaller. *Moore's Law: Past, Present, and Future*. IEEE Spectr., vol. 34, no. 6, pages 52–59, June 1997. (Cited on page 4.)

[Serot 2008] Jocelyn Serot and Joel Falcou. *Functional meta-programming for parallel skeletons*. In Computational Science–ICCS 2008, pages 154–163. Springer Berlin Heidelberg, 2008. (Cited on page 23.)

[Sheard 2002] Tim Sheard and Simon Peyton Jones. *Template meta-programming for Haskell*. In Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, pages 1–16. ACM, 2002. (Cited on page 23.)

[Siek 2002] Jeremy G. Siek, Lie-Quan Lee and Andrew Lumsdaine. The boost graph library - user guide and reference manual. C++ in-depth series. Pearson / Prentice Hall, 2002. (Cited on page 22.)

[Siek 2005] Jeremy G. Siek and Andrew Lumsdaine. *Essential language support for generic programming*. In PLDI, pages 73–84, 2005. (Cited on page 21.)

[Smith 1981] T. Smith and M. Waterman. *Identification of common molecular subsequences*. Journal of Molecular Biology, vol. 147, pages 195–197, 1981. (Cited on page 39.)

[Spinellis 2001] Diomidis Spinellis. *Notable design patterns for domain-specific languages*. Journal of Systems and Software, vol. 56, no. 1, pages 91 – 99, 2001. (Cited on page 23.)

[Standard 2014] C++ Standard. *The Callable Concept*. `http://en.cppreference.com/w/cpp/concept/Callable`, 2014. (Cited on page 27.)

[Stepanov 1995a] Alexander Stepanov and Meng Lee. *The Standard Template Library*. Rapport technique 95-11(R.1), HP Laboratories, 1995. (Cited on page 19.)

[Stepanov 1995b] Alexander Stepanov and Meng Lee. *The Standard Template Library*. Rapport technique 95-11(R.1), HP Laboratories, 1995. (Cited on page 64.)

[Suilen 2006] W. Suilen. *The BSP on MPI Library*, 2006. (Cited on page 11.)

[Sutter 2005] Herb Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. Dr. Dobb's Journal, vol. 30, no. 3, pages 202–210, 2005. (Cited on page 4.)

[Sutton 2011] Andrew Sutton and Bjarne Stroustrup. *Design of Concept Libraries for C++*. In SLE, pages 97–118, 2011. (Cited on page 21.)

[Tan 2014] Antoine Tran Tan, Joel Falcou, Daniel Etiemble, Hartmut Kaiser*et al.* *Automatic Task-based Code Generation for High Performance Domain Specific Embedded Language*. In HLPP 2014, 2014. (Cited on page 69.)

[The C++ Standards Committee 2011] The C++ Standards Committee. *ISO/IEC 14882:2011, Standard for Programming Language C++*. Rapport technique, ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee, 2011. http://www.open-std.org/jtc1/sc22/wg21. (Cited on page 74.)

[Unruh 1994] Erwin Unruh. *Prime number computation*. ANSI X3J16-94-0075/ISO WG21-462, 1994. (Cited on page 23.)

[Valiant 1990] Leslie G. Valiant. *A Bridging Model for Parallel Computation*. Commun. ACM, vol. 33, no. 8, pages 103–111, 1990. (Cited on page 10.)

[Vandevoorde 2002] David Vandevoorde and Nicolai M. Josuttis. C++ templates. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. (Cited on page 23.)

[Veldhuizen 1995] Todd L. Veldhuizen. *Expression templates*. C++ Report, vol. 7, no. 5, pages 26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman. (Cited on page 23.)

[Veldhuizen 1998] Todd L. Veldhuizen and Dennis Gannon. *Active Libraries: Rethinking the roles of compilers and libraries*. In In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98. SIAM Press, 1998. (Cited on page 22.)

[Yarkhan 2011] Asim Yarkhan, Jakub Kurzak and Jack Dongarra. *QUARK Users Guide*. Rapport technique, Technical Report April, Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tenessee, 2011. (Cited on page 74.)

[Ye 2013] H Ye, Lionel Lacassagne, Joël Falcou, Daniel Etiemble, L Cabaret and O Florent. *High level tranforms toreduce energy consumption of signal and image processing operators*. In Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013 23rd International Workshop on, pages 247–254. IEEE, 2013. (Cited on page 80.)

[Yelick 1998] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay,

Phil Colella and Alex Aiken. *Titanium: a high-performance Java dialect.* Concurrency: Practice and Experience, vol. 10, no. 11-13, pages 825–836, 1998. (Cited on page 13.)